ELSEVIER Science of Computer Programming 33 (1999) 29–85

Science of Computer Programming

Undulant-block elimination and integer-preserving matrix inversion

David S. Wise*

Computer Science Department, Indiana University, Bloomington, IN 47405-4101, USA

Communicated by P. Cousot; received 30 August 1995; accepted 5 February 1998

Abstract

A new formulation for LU decomposition allows efficient representation of intermediate matrices while eliminating blocks of various sizes, i.e. during "undulant-block" elimination. Its efficiency arises from its design for block encapsulization, implicit in data structures that are convenient both for process scheduling and for memory management. Row/column permutations that can destroy such encapsulizations are deferred. Its algorithms, expressed naturally as functional programs, are well suited to parallel and distributed processing.

A given matrix A is decomposed into two matrices (in the space of just one), plus two permutations. The permutations, P and Q, are the row/column rearrangements usual to complete pivoting. The principal results are L and U', where L is properly lower quasi-triangular; U' is upper quasi-triangular with its quasi-diagonal being the inverse of that of U from the usual factorization ($PAQ = (I - L)U$), and its proper upper portion identical to U. The matrix result is $L + U'$. Algorithms for solving linear systems and matrix inversion follow directly.

An example of a motivating data structure, the quadtree representation for matrices, is reviewed. Candidate pivots for Gaussian elimination under that structure are the subtrees, both constraining and assisting the pivot search, as well as decomposing to independent block/tree operations. The elementary algorithms are provided, coded in HASKELL.

Finally, an integer-preserving version is presented replacing Bareiss's algorithm with a parallel equivalent. The decomposition of an integer matrix A to integer matrices $\bar{L}$, $\bar{U}'$, and $d = \det A$ follows $L + U'$ decomposition, but the follow-on algorithm to compute dA^{-1} is complicated by the requirement to maintain minimal denominators at every step and to avoid divisions, restricting them to necessarily exact ones.

AMS Classification: primary 65F05; secondary 68Q22; 65F50

CCS Categories and Subject Descriptors: G.1.3 [Numerical Analysis]: Numerical Linear Algebra —linear systems, matrix inversion, sparse and very large systems; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors); E.1 [Data Structures]—trees; D.1.1 [Programming Techniques]: Applicative (Functional) Programming; 1.1.2 [Symbolic and Algebraic Manipulation]: Algorithms—algebraic algorithms, analysis of algorithms; F.2.1

* E-mail: dswise@cs.indiana.edu. Research reported herein was sponsored, in part, by the National Science Foundation under Grant Number DCR 90-027092.

0167-6423/99/$ – see front matter
PII: S0167-6423(98)00005-7

[Analysis ofAlgorithms and Problem Complexity]: Numerical Algorithms and Problems —computations on matrices.
General Term: Algorithms

Keywords: Gaussian elimination; Block algorithm; *LU* factorization; Quadtree matrices; Quaternary trees; Pivoting; Exact arithmetic

0. Introduction

> You can measure a programmer's perspective by noting his attitude on the continuing vitality of FORTRAN. [18, ¶42]

Problems from linear algebra, like solving linear systems and inverting matrices, occupy very important roles in the development of computing. Direct and indirect solutions have consumed so many cycles over the years that support for straightforward decomposition algorithms can be found deep in programming language designs, in compilers, in operating systems, and in processors, themselves. Indeed, any measurement of performance in "flops" confirms their impact. Their solutions, thereby, have become bellwethers on the current frontier of computing, whose performance is noticed even by those who never use them.

0.1. The audience

This paper is intended to reach a wide readership in computing research. Diverse fields of your interests might include matrix algebra, symbolic algebra, scientific computation, analysis of algorithms, programming languages, and multiprocessing. An explicit goal of this paper is to engage colleagues, active in these fields, in the refinement and testing of the general approach presented here. I hope that narrow reactions, enforcing the traditional practices in any one of these fields – to the exclusion of others' – can be stifled. Not so long ago "traditional practices" hardly existed in computing research, as we stood together before the bar of Science. Rapid progress first divided and then isolated us, so that the practices of one field now distinguish it as much from others within computing, as from other scientific disciplines. Such a trend is not only scientifically but also politically foolish; we are all colleagues learning how to compute. We must better share problems, solutions, styles, techniques, and philosophy.

Depending on your background, therefore, you may find some parts of this paper dull while straining to master others. An expert in matrix algebra should be able to scan quickly Sections 2.3 and 2.4 on block-Gaussian elimination, but may stumble on the HASKELL code in Sections 1.4 and 3.1; in contrast, an expert in programming languages could have the exactly opposite difficulty. That colleague will also have little trouble understanding the parallelism implicit in `mapping` functions across subtrees, where a master of pipelined architectures might, at first, find such limited control

meaningless. A numerical analyst might seek details on timings or pivot selection (to appear in a later paper) of the floating-point algorithms in Section 3, without allowing for the locality of their memory use or the generality of the representation: uniform for sparse and dense matrices and amenable to hybrids. Any concerns over stability would contrast sharply with the exact arithmetic in Section 4, of interest to a symbolic algebraist. Although these two colleagues might appear to be brethren to a student, say, of analysis of algorithms, their literature is disjoint; but this paper addresses all.

Strange are the divisions in our discipline! These broad ideas should be debated by computing researchers communicating with – rather than at – one another.

0.2. The approach

This paper shuns the traditional row-major or column-major representation of matrices in favor of a block-oriented one, and revisits the direct algorithm, Gaussian elimination (GE), from the perspective of block decomposition. While a specific representation for matrices is presented, the algorithms, themselves, are presented independently of such a representation; they are suited to whatever block decomposition is best for your hardware or operating system. Most importantly, the order of a block-operand is here allowed to vary irregularly from one step to the next; this so-called "undulant" blocking is the major distinction between these algorithms and others' [16, 7] that impose a uniform order.

Three interrelated contributions can be found in this paper.

- The collection of block algorithms for undulant-block decomposition along with associated sequels for solving a linear system and for inverting a matrix. A significant feature is deferring of all permutations associated with pivoting, which are presumed to be expensive under any block-structured representation.
- The quadtree representation for matrices that unifies the representation for sparse and non-sparse matrices and distributes complete pivoting across elimination steps, using local computation exclusively.
- The integer-preserving versions of the decomposition and sequela for use with exact arithmetic and symbolic manipulation.

Non-trivial blocking uses hierarchical [15, 13] and distributed memory more effectively than row-based methods. For example, with cache that holds $3m^2$ scalars the multiplication of two $m^2 \times m^2$ matrices does more work with less communication when the cache load is three $m \times m$ blocks rather than full rows and columns of length m^2. Before fetching from memory, the second allows only $2m^2$ scalar multiplications to complete two entries in the product; but the first readily computes m products added to each of m^2 entries. Under parallel processing, block manipulation reduces the granularity of both scheduling and communication [6]. Such improvement from block operations is well known; matrix–matrix operations are commonly called Level 3 (BLAS) [8], with matrix–vector operations rated Level 2.

Statically sized blocks are used by existing block algorithms for GE under one of two philosophies. They can be used without constraint in cases when it is "safe" not to pivot (i.e. elimination of a block on the main diagonal is assured), for example when the problem is diagonally dominant or symmetric positive-definite. Alternatively, they can be used heuristically, against the remote possibility of collapse when a non-singular residual matrix has no non-singular, stable-pivot block of the required size and orientation. Upon such a failure, the partial solution is usually abandoned and the problem reordered to be solved again afresh, even though scalar pivoting – the simplest of undulant-block strategies – would complete that partial solution.

An enabling concession under undulant-block pivoting is that only certain subblocks of certain orders can be candidates for elimination in a single step. The specific restrictions depend on the data structure. Since the representation of interest here is the quaternary tree decomposition, the pivot candidates coincide with proper subtrees whose orders increase by doubling. Moreover, the tree structure, itself includes local information ("decorations" on the nonterminal nodes) for summary information that guides pivoting. Each decoration is computed and stored locally during elimination, and used only when that block again participates in an elimination step; intervening steps that do not touch that block cannot invalidate its decoration.

0.3. The paper

The remainder of this paper is in six parts. The first reviews the quadtree representation for numeric and permutation matrices, and a HASKELL implementation of it. The second presents the generalization of LU decomposition: $L+U'$ decomposition, and associated algorithms for solving linear systems and matrix inversion that are designed for undulant elimination under any block-oriented matrix representation. The third section marries the quadtree representation to these algorithms, including typical codes and subsections on embedding padding within the permutations and decorating the trees for pivoting. Section 4 presents the exact-arithmetic analogs of these algorithms, and Section 5 anticipates improvements from ordering the basis. Finally, the last section offers conclusions.

1. Quadtree representation

> Symmetry is a complexity-reducing concept (co-routines include subroutines); seek it everywhere. [18, ¶6]

1.1. Arrays as trees

The quadtree representation of matrices [21] motivated the algorithms to follow. Conceptually, it decomposes full matrices as balanced quaternary trees. Sparse matrices are accommodated by providing a distinguished representation for an entirely zero

submatrix, essentially a *null* pointer (cf. hypermatrices [9]). The same convention makes it easy to handle a matrix whose order, n, is not a power of two by padding with zero blocks to order $2^{\lceil \lg n \rceil}$.

This paper is written as if the leaves of these tree were scalars – 1×1 blocks – but this convention is not essential. Indeed, the domain of time-dependent differential equations suggests a 3×3 leaf, and cache architectures encourage a 4×4. Within such leaf-submatrices, sequential allocation/indexing and conventional algorithms could take advantage of existing architecture, which has been tuned to them. The resulting strategy – likely the best one – is, therefore, hybrid.

Block-as-subtree makes it easy to communicate submatrices to remote processes; instead of communicating the contents of a block or even the indices that delimit it, only a single pointer need be communicated at rendezvous, regardless of the size of the submatrix. Its content can be copied later, asynchronously. Furthermore, this block structure for matrices makes it natural to restrict the pivot search in GE to blocks that are coincident with subtrees, constraining the search, and accelerating both it and the elimination step to follow. For instance, one can identify $(n-1)^2$ contiguous 2×2 blocks in an $n \times n$ matrix, but only $n^2/4$ of them are subtrees, and so admissible as pivots.

Postulate. *A d-dimensional* array *is represented as a* 2^d*-ary tree.*

Data structure (Binary tree representation of vectors). *A* vector *of size* 2^p, *represented as a binary tree of depth* p, *is*

- *homogeneously zero and represented as* 0;
- *represented by the appropriate non-zero scalar when* $p = 0$;
- *otherwise represented as a pair of subvectors*, (north, south), *each of which is of size* 2^{p-1}, *at least one of which is non-zero.*

Data structure (Quadtree representation of matrices). *A* matrix *of order* 2^p, *represented as a quaternary tree of depth* p, *is*

- *homogeneously zero and represented as* 0;
- *represented by the appropriate non-zero scalar when* $p = 0$;
- *otherwise represented as a quadruple of submatrices*, (northwest, northeast, southwest, southeast), *each of which is order* 2^{p-1}, *and at least one of which is non-zero.*

Definition. A *stripe*[1] is a set of adjacent rows in a matrix. A *colonnade* is a set of adjacent columns.

1.2. Indexing

Definition. The Ahnentafel index [5] of an entire vector is 1. If i is the Ahnentafel index of a subvector, then the Ahnentafel index of its north half is $2i$, and the Ahnentafel index of its south half is $2i + 1$.

[1] As in "Stars and Stripes" with apologies to tigers and zebras.

This is the familiar "level order" indexing of a binary tree, where the 2^l nodes at Level l are indexed left-to-right with $(l+1)$-bit integers from 2^l through $2^{l+1}-1$.

Theorem 1.1. *A node with Ahnentafel index i occurs at Level $\lfloor \lg i \rfloor$ in the binary tree.*

Definition. The function A maps the triple $\langle i,k,n \rangle$ to the Ahnentafel index corresponding to that of a subvector/subtree of length k starting i positions from the north end of a vector of order n.

Necessarily n is a power of two, $0 \leqslant i < n$, and $0 < i + k \leqslant n$; for the subvector to coincide with a subtree both $k|n$ and $k|i$.

Theorem 1.2. $\mathrm{A}(i,k,n) = (n+i)/k$.

Proof. This recurrence suffices as an intermediate step:

$$\mathrm{A}(0,n,n) = 1; \quad \mathrm{A}(0,k,2n) = 2\mathrm{A}(0,k,n); \quad \mathrm{A}(i+k,k,n) = \mathrm{A}(i,k,n) + 1. \qquad \square$$

Theorem 1.3. *Whenever a vector is built from a north half and a south half, each already Ahnentafel indexed, then the Ahnentafel index of the whole new vector is* 1, *and the relative index of any node in the north or south half is $i + 2^{\lfloor \lg i \rfloor}$ (respectively $i + 2^{\lfloor \lg i \rfloor + 1}$) where i was its index in the half that is north or south (respectively where i was its index in the south vector).*

Proof. Consider Level $l+1$ in the tree that is the new vector, corresponding to Level l in each of the preindexed halves. It is indexed consecutively beginning at 2^{l+1}, with southern indices beginning at $2^{l+1} + 2^l$. The corresponding Level l in both preindexed vectors was indexed consecutively beginning at 2^l, so adding 2^l to respective north indices and 2^{l+1} to respective south indices at that level generates the desired, consecutive indexing at this level in the new vector. $\square$

In the algorithms to follow, however, all indices will be reverse-Ahnentafel indices; see Fig. 1.

Definition. The *reverse-Ahnentafel index* of an entire vector of length 2^p is 1. If i is the reverse-Ahnentafel index of its subvector of length 2^{p-l} (where $l = \lfloor \lg i \rfloor$), then the reverse-Ahnentafel index of its north half (of order 2^{p-l-1}) is $i + 2^l$, and the reverse-Ahnentafel index of its south half is $i + 2^{l+1}$.

Theorem 1.4. *Except for the most significant bit, the sequence of bits in the binary representations of the Ahnentafel index and the reverse-Ahnentafel index of any node are the reverse of one, another.*

Proof. By simple induction on l, the level of the node in the binary tree, which is also the length of the reversible sequence. $\square$

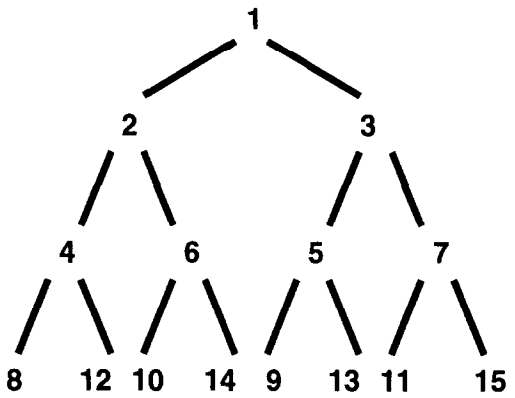

Fig. 1. Reverse-Ahnentafel indexing in a vector of size 8.

Theorem 1.5. *Whenever a vector is built from a north half and a south half, each already reverse-Ahnentafel indexed, then the reverse-Ahnentafel index of the whole new vector is* 1, *and the relative index of any node in the north or south half is* $2i$ (*respectively* $2i+1$), *where* i *was its index in the half that is north or south, respectively.*

Proof. Theorem 1.3 establishes the duality between Ahnentafel indexing and reverse-Ahnentafel indexing. Consistently with that duality, Theorem 1.4 uses the computations from the definition of Ahnentafel indexing to reindex north and subtrees under reverse-Ahnentafel indexing, just as Theorem 1.2 uses the computations from the definition of reverse-Ahnentafel indexing to solve the same problem under Ahnentafel indexing. □

Definition. $\mathrm{R}(1)=1$. For integer $k>0$ and either $b=0$ or $b=1$, $\mathrm{R}(2k+b)=\mathrm{R}(k)+2^{\lfloor \lg k \rfloor + b}$.

Corollary 1.1. *The function* R *maps an Ahnentafel index to its reverse-Ahnentafel index, and vice versa.*

Thus, (R · A), succinctly RA, computes the reverse-Ahnentafel index of a cartesian-index triple, $\langle i,k,n \rangle$, for a subvector of length k starting i positions from the north end of a vector of order n.

The algorithm in the next sections will frequently construct indices bottom-up and discharge them top-down; the computations from Theorem 1.4 – as well as their inverses – will be used often. Using binary arithmetic, we can easily build such indices bottom-up, using doubling and addition at each node, and discharge them top-down, descending the binary tree simply using integer `quotient_remainder` on 2 at each level.

It is because these computations occur repeatedly during Œ, that reverse-Ahnentafel indexing is used in this paper.[2]

Definition. The meaning of an *index into a matrix*, $\langle i,j\rangle$, is defined when $\lfloor \lg i \rfloor = \lfloor \lg j \rfloor$.
- The pair $\langle 1,1\rangle$ indexes the entire matrix.
- The pair $\langle 2i,2j\rangle$ indexes to the $\langle i,j\rangle$ entry in the northwest submatrix;
- the pair $\langle 2i,2j+1\rangle$ indexes to the $\langle i,j\rangle$ entry in the northeast submatrix;
- the pair $\langle 2i+1,2j\rangle$ indexes to the $\langle i,j\rangle$ entry in the southwest submatrix;
- the pair $\langle 2i+1,2j+1\rangle$ indexes to the $\langle i,j\rangle$ entry in the southeast submatrix.

Burton and Kollias [4] use an Ahnentafel-like indexing generalized from binary to quaternary trees, without convenient stripe/colonnade indexing. The only stripes and colonnades of interest in quadtree matrices will be those that have Ahnentafel indices. That is, the first index in the pair indexing into a matrix identifies a stripe, and the second identifies a colonnade.

1.3. Permutation matrices

Notation. *I denotes the* identity *matrix of any order. Similarly,* 0 *denotes the* zero *matrix of any order.*

Data structure. *A* zero–one matrix *is a matrix of order 2^p that is represented as*
- *the homogeneously zero matrix, represented as* 0;
- *the identity matrix, represented as I;*
- *otherwise represented as a quadruple of submatrices,* (northwest, northeast, southwest, southeast), *each of which is order 2^{p-1} and each zero–one.*

Data structure. *A* quadtree-permutation matrix *is a zero–one matrix that satisfies the two constraints:*
- *Every stripe either contains exactly one I entry that spans the stripe, or it can be split into two stripes, each of which has this property, recursively.*
- *Every colonnade either contains exactly one I entry that spans the colonnade, or it can be split into two colonnades of equal size, each of which has this property.*

These two qualifications establish the "eight-rooks problem" orientation of I entries in permutation matrices: each indexable stripe and colonnade is spanned by, in aggregate, one I entry. Alternatively stated: every row and every column has an I entry, and if indices, $\langle i,j\rangle$ and $\langle m,n\rangle$, of two I blocks satisfy either equation $m = k2^{\lfloor i \rfloor} + i$ or

[2] Reverse-Ahnentafel indexing, like floating-point numbers and even the quadtree representation, itself, is an internal representation that speeds computation. All three are isomorphic to alternative representations that are more easily read by humans, but translations often are computationally difficult [19]. Such translation never occurs during routine computation, however, and are overlapped with trudging input/output whenever they become necessary.

Table 1
Costs of patterned and unpatterned matrices as quadtrees

Pattern	Space	Expected path
Full	$\frac{4}{3}n^2$	$\lg n + 1$
Symmetric	$\frac{2}{3}n^2$	$\lg n + 1$
Hankel/Toeplitz	$4n$	$\lg n + 1$
Triangular	$\frac{2}{3}n^2$	$\frac{1}{2}\lg n + 1.5$
fFT permutation	$\frac{1}{2}n \lg n$	$\frac{1}{2}\lg n + 1.34$
Random permutation	$\frac{1}{2}n \lg n$	$\frac{1}{2}\lg n + 0.9$
Diagonal	$2n$	2
Tridiagonal	$6n$	3.34
Pentadiagonal	$8n$	3.34
Heptadiagonal	$11n$	3.34
Enneadiagonal	$13n$	3.34
Shuffle permutation	$3n$	3
Zero	0	0

$i = k2^{\lfloor m \rfloor} + m$ for integer k, then $k = 0$ and $j = n$; that is, if one spans the stripe containing another, then they coincide. Similarly, if either $j = k2^{\lfloor n \rfloor} + n$ or $n = k2^{\lfloor j \rfloor} + j$ then $k = 0$ and $i = m$.

Definition. A permutation is said to be of "even" parity if, expressed as a matrix, it has determinant $+1$. It is of "odd" parity when its determinant is -1.

Thus, a permutation's parity is just the sign of its determinant, necessarily of unit magnitude.

Corollary 1.2. *No non-terminal node in a quadtree representation has* 0 *as all four of its quadrants. Similarly, no representation of a quadtree-permutation matrix has both northeast and southwest quadrants* 0 *while both its northwest and southeast are simultaneously either* 0 *or I.*

Table 1 summarizes space and access-time asymptotes extracted from the analytic results of Wise and Franco [23]. They show how familiarly patterned matrices are uniformly represented in expectedly shrinking space, albeit with proportional overhead beyond case-specific data structures. The expected path here reflects the cost to access a random $[i, j]$ element of a matrix, from the root of the entire tree.

Although Table 1 shows that this measure also decreases with patterning, good quadtree algorithms will *not* probe these structures from their roots; instead the recurrences of these algorithms apply locally to deeper subtrees. The next section shows how addition and multiplication, for instance, decompose to independent, parallel processes

on subtrees. The quadtree structure encourages such divide-and-conquer algorithms, which descend into their operand-trees, and probe to leaves from levels deep in the tree, from twigs near the leaves.

1.4. Ring algorithms

HASKELL [12] is the programming language used in this paper. HASKELL is the international functional-programming language and, therefore, its code is devoid of any sequentiality except that implicit in the dependence of one result upon another. The structure of the code builds on the declarations of its strong data types; a typical line of code matches these types to define one case of a function's definition. To the left of an "equals sign" is a pattern keyed on unique Constructors (upper case) and binding local parameters (lower case); to its right is the definition of a value for the specified case.

Another feature of HASKELL, well used here, is its mapping functionals, that distribute a single function across one or more aggregate structures. LISP calls them `mapcar` or `mapcar2`; SCHEME calls it `map`. HASKELL's strong typing encouraged the strange menagerie of `map` for distributing across a single aggregate, `zipWith` across two, and `zipWith3`, etc., across more. It is presumed here that this family of functionals applies to homogeneous tuples and structures, as well to as to lists; this extension is very important for identification at compile time of parallelism of fixed degree.

A HASKELL declaration for a `Matrx` whose elements are of type `a` appears in Fig. 2. The constructed data type, declared in the first three lines, unifies the matrices from Section 1.3 and which specifies four alternatives:

- 0,
- a 1×1 scalar matrix of type `a`,
- a tuple of four quadrants,
- or I (used only for permutation matrices).

Fig. 2 also exhibits code for ring operations on this type that take advantage of the algebraic properties of 0 and I; HASKELL provides syntactic overloading of `+`. `-`, and `*` via declaration of matrices and vectors as instances of its `Num` class; of course, any underlying scalar type `a` must also be in `Num` so these operators are defined over it a priori. (Practical candidates for `a` include `Integer`, `Rational`, `Float`, and `Double`.)

The eight-rooks layout of permutation matrices assures that multiplication in Fig. 2 on a quadtree-permutation matrix never adds two non-zero terms, especially `IdentM`; the annihilator and identity properties of 0 and I there prevent it. Comments are initiated by – and extend to end-of-line.

Parallel realization of such code depends upon proper treatment of two of its features. First, process dispatch is to be associated with function invocation *top-down* insofar as the supply of processors permits. Second, parallel dispatch is suggested by any collateral argument evaluation, but most importantly at the `zipWith` function mappings. These mappings are important for scheduling because they identify subprocesses of approximately uniform load and importance, balancing the schedule; ordinary collateral

```
type Quadrants a = (Matrx a, Matrx a, Matrx a, Matrx a)
data Matrx  a =   ZeroM | ScalarM a | Mtx  (Quadrants a)
                  | IdentM          --used only in permutations.
instance (Num a) => Num (Matrx a) where
    fromInteger 0         = ZeroM
    fromInteger 1         = IdentM
    negate ZeroM          = ZeroM
    negate (ScalarM x)    = ScalarM (negate x)
    negate (Mtx quads)    = Mtx (map negate quads)
    x         + ZeroM     = x             --NB: In the case of a direct sum,
    ZeroM     + y         = y             --the base case must add to ZeroM.
    ScalarM x + ScalarM y = case x+y of z |z==fromInteger 0  -> ZeroM
                                          |otherwise         -> ScalarM z
    Mtx x     + Mtx y     = normalize (zipWith (+) x y)
    x         - ZeroM     = x             --Subtraction won't handle IdentM.
    ZeroM     - y         = negate y
    ScalarM x - ScalarM y = case x-y of z |z==fromInteger 0  -> ZeroM
                                          |otherwise         -> ScalarM z
    Mtx x     - Mtx y     = normalize (zipWith (-) x y)
    ZeroM     * _         = ZeroM
    _         * ZeroM     = ZeroM
    IdentM    * y         = y        --NB:  Multiplication on IdentM works.
    x         * IdentM    = x
    ScalarM x * ScalarM y = ScalarM (x*y)
       --Except with infinitesimal floats:  case x*y of  0->0; z->ScalarM z
    Mtx x     * Mtx y     = normalize (zipWith (+)
                                (zipWith (*) (colExchange x)(offDiagSqsh y))
                                (zipWith (*)       x        (prmDiagSqsh y))
                                )
                --If stripe of x or colonnade of y is like a permutation's,
                --   then the respective, nested sum is a direct sum.

colExchange (nw,ne,sw,se) = (ne,nw,se,sw)  --Quadrant permutation.
prmDiagSqsh (nw,ne,sw,se) = (nw,se,nw,se)  --Read "primary-diagonal squash"
offDiagSqsh (nw,ne,sw,se) = (sw,ne,sw,ne)  --Read "off-diagonal squash".
normalize (ZeroM,ZeroM,ZeroM,ZeroM) = ZeroM
normalize quads                     = Mtx quads

data (Num a) => Vectr a =   ZeroV | ScalarV a | Vec (Vectr a)(Vectr a)
instance (Num a) => Num (Vectr a) where
    fromInteger 0 = ZeroM
    fromInteger s = ScalarV s
    x         + ZeroV     = x
    ZeroV     + y         = y        --NB:   A direct sum stops here!
    ScalarV x + ScalarV y = case x+y of z |z==fromInteger 0  -> ZeroV
                                          |otherwise         -> ScalarV z
    x         + y         = case (zipWith (+) x y) of
                                Vec ZeroV ZeroV = ZeroV
                                z               = z
```

Fig. 2. HASKELL code for quadtree matrices as instance of Num.

argument evaluation carries no implication about balance. Language designers should make these mapping functions more convenient, and programmers should be using them more, close to the "root" of their programs' structure, dividing to (and conquering) tasks of about the same size to balance the load on future multiprocessors.

For instance, the matrix addition of two dense 4096×4096 matrices using 16 processors would create exactly 16 subprocesses dispatched once, each adding 1024×1024 submatrices; thus, the overhead for each process dispatch and recovery would be amortized against a million scalar additions. (With storage management and processor management working together, these subprocesses might require only local data: residing together on the same processor/page/cache-line.) Of course, if the matrix were sparse, some of these processes might complete sooner than others, and their processors could be redispatched to help the remainder: say on residual 512×512 additions. An alternative reading – that the processes are dispatched bottom-up on 16 scalar additions at a time, but a million times – is possible, though silly.

1.5. Operation counts

The results entitled "OpCount" below present bounds on the *uniprocessor* behavior of these algorithms [1]. They measure the total number of multiplications or divisions, the traditional metric for linear systems, even though it can be misleading for multiprocessing. Like theorems or corollaries each can be established from the relevant code and previous opCounts, usually by inspection.

Conflicts in accessing shared data is a greater constraint on multiprocessors than serial piping through *the* multiplier, simply because there can be many multipliers. If 4×4 leaves of a quadtree were stored serially, moreover, the time to fetch and to multiply them will be little different from the time to fetch and to multiply two scalars; effects from latency and caching make the actual multiplication time trivial [6]. However, only $(n/4)^3$ block multiplications of the first sort are necessary to multiply $n \times n$ matrices, compared to n^3 of the second. So the coefficients that appear these uniprocessor-operation counts may not themselves be useful, except for relative comparison.

OpCount 1.1. *The multiplication in Fig.* 2 *of two full* $n \times n$ *matrices requires at most* n^3 *scalar multiplications.*

OpCount 1.2. *The multiplication in Fig.* 2 *of a full and a triangular* $n \times n$ *matrix requires at most* $(n^3 + n^2)/2$ *scalar multiplications.*

OpCount 1.3. *The multiplication in Fig.* 2 *of two triangular* $n \times n$ *matrices requires at most* $(n^3 + n^2)/3$ *scalar multiplications.*

An alternative to the product algorithm in Fig. 2 is Strassen's algorithm [17, Section 1.3.8], for which quadtrees are the ideal structure. It is not included here both for simplicity and because its pre-additions eradicate sparseness [21, 23] that might accelerate later multiplications. If it were included, however, all the n^3 factors in these counts could be replaced by $n^{2.81}$ with the same coefficients. That fact is of little immediate interest, however, since the purpose of the counts here is to characterize the relative

costs of different algorithms, all of the same polynomial order. That *relative* cost is reflected by these coefficients.

2. $L + U'$ decomposition

> Functions delay binding; data structures induce binding. Moral: Structure data late in the programming process. [18, ¶2]

An undulant version of LU decomposition is presented independent of the quadtree representation of matrices. While the theory here is not new, this presentation replaces the traditional row/column decomposition of matrices with arbitrary-block decomposition. Since no specific data structure is used, the algorithms developed here can be used with any block-oriented matrix representation. The example and its Ahnentafel indexing scheme, however, anticipate their use on quadtree matrices in later sections.

Notation. *Upper-case Greek letters denote lists of integers.*

2.1. Quasi-triangular matrices

The indexing in the next few definitions is the ordinary cartesian row/column indexing usual for scalar entries in a matrix. Familiarity with the triangular LU decomposition via scalar elimination is assumed [17, Section 3.2].

Definition. A matrix A is properly-lower (or -upper) triangular if $a_{i,j} = 0$ for $i \leqslant j$ (respectively, $i \geqslant j$).

Corollary 2.1. *If L is properly-lower triangular, then $I - L$ is unit-lower triangular.*

OpCount 2.1. *The $n \times n$ product in Fig. 2 of a full matrix and a unit- or properly triangular matrix requires at most $(n^3 - n^2)/2$ scalar multiplications.*

OpCount 2.2. *The $n \times n$ product in Fig. 2 of an upper-triangular matrix, and a unit- or properly lower triangular matrix requires at most $(n^3 - n^2)/3$ scalar multiplications.*

Definition. Two matrices, A and B, are said to be *disjoint* if for all i, j either $a_{i,j} = 0$ or $b_{i,j} = 0$.

Definition. A square matrix is *quasi-diagonal* if it has square submatrices (cells) along its main diagonal with its remaining elements equal to zero.

The older term, "quasi-diagonal" [11, Section 1.5], is used, instead of "block-diagonal" [17, Section 1.3.1], to emphasize that the diagonal blocks are square. The square cells, of varying or *undulating* size will coincide with eliminated blocks, in the

order in which they are eliminated; since the cells' sizes do undulate, their sizes must be recorded.

Notation. *The integer k is used globally to indicate the number of elimination steps, and locally to indicate the size of a specific diagonal block.*

If D is an $n \times n$ quasi-diagonal matrix with k non-zero blocks then, following the block decomposition, one could partition the basis of the underlying vector space, decomposing it into k mutually complementary subspaces, of independent sizes, together spanning the entire space. Thus, problems on D can be decomposed into k small, independent problems: one in each subspace and each, perhaps, a different size.

Definition. A square matrix is lower (upper) *quasi-triangular* if it has square cells along its main diagonal with its remaining elements above (respectively, below) that diagonal equal to zero. It is *properly* quasi-triangular when it the diagonal cells, themselves, are also zero.

The properly lower quasi-triangular matrix, L, associated below with a quasi-diagonal matrix, D (or upper quasi-triangular U), has zero submatrices exactly where D (or U) necessarily has non-zero matrices. Therefore, the aforementioned decomposition on the vector space underlying D can be applied to $I - L$, as well. The same can be said for associated upper-triangular matrix, U, that includes D.

Definition. An $L + U'$ *decomposition* of A, non-singular of order n, is the quintuple, $\langle d, L + U', \Omega, P, Q \rangle$ where

- $d = \det A$;
- P and Q are permutation matrices;
- $L + U', P, Q$ are matrices of the same order as A;
- The sum of the elements of Ω is the order of A;
- Ω is a list of the orders prescribing D, the quasi-diagonal portion of $L + U'$;
- L is properly lower-triangular portion of $L + U'$ whose bounding quasi-diagonal is described by the entries in Ω;
- U' is upper quasi-triangular, using the same boundary, including D and disjoint from L;
- $U = U' - D + D^{-1}$;
- $PAQ = (I - L)U$.

Fig. 3 sketches the different images of triangular LU decomposition, and quasi-triangular $L + U$ and $L + U'$ decompositions. It is trivial to separate $(L + U')$ into unit-lower and upper-triangular matrices, $I - L$ and U', but this is never necessary. P and Q are the row and column permutations usual to complete pivoting.

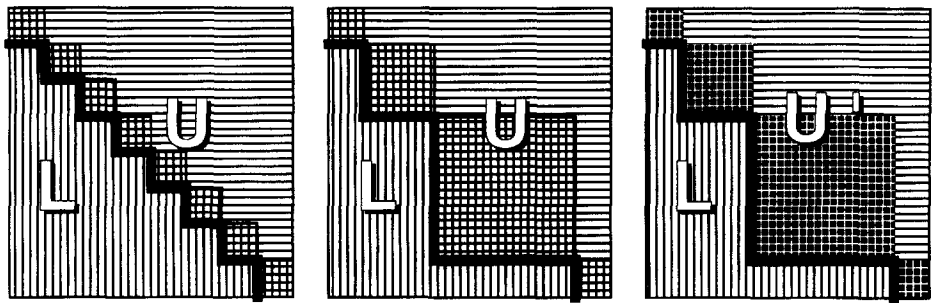

Fig. 3. 8×8 triangular $L + U$, versus quasi-triangular $L + U$, versus $L + U'$.

2.2. Permutations as lists of indices

Permutations P and Q can be represented instead by a sequence of reverse-Ahnentafel indices. This convention is not essential either to the definitions above or to the undulant decomposition algorithms presented later in this section. It is, however, our goal in Section 3.

Definition. The *series encoding* Π of a row permutation P (respectively, Ψ of a column permutation Q) is a list of reverse-Ahnentafel indices locating I entries in otherwise-zero vectors that are rows (columns) of the permutation matrix, P (Q); each entry i expands to a stripe of breadth $2^{\lfloor \lg i \rfloor}$ the order of each 0 or I entry in that vector.

When applied to P, Q in the decomposition, this definition generates Π, Ψ of the same length as Ω, with the entries in Ω being the length of those stripes. Following this theorem is an example that is developed throughout the remainder of this paper.

Theorem 2.1. *Let A be of order 2^p, and Ω, Π, Ψ be sequences resulting from decomposing A. Then*

$$\mathrm{map}(\lambda i.2^{p-\lfloor \lg i \rfloor})\Pi = \mathrm{map}(\lambda j.2^{p-\lfloor \lg j \rfloor})\Psi = \Omega.$$

Example. Pivoting on a 4×4 matrix yields $\Pi = \langle 5,2,7 \rangle$ and $\Psi = \langle 4,3,6 \rangle$; what do these sequences mean?

Worth checking is (Theorem 2.1) that applying $(\lambda i.2^{2-\lfloor \lg i \rfloor})$ to each index in these sequences yields the same $\Omega = \langle 1,2,1 \rangle$, suggesting that the rows (columns) of P (Q) are to be clustered in stripes (colonnades) of one, two, and one reading top-to-bottom (left-to-right). Each is a vector of length four, two, and four, respectively. The first three levels of the tree in Fig. 1, show how $\Pi \mapsto P$ and $\Psi \mapsto Q$; so, these sequences

identify the permutations:

$$P = \left(\begin{array}{cccc} 0 & 0 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right), \qquad Q = \left(\begin{array}{c|cc|c} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right).$$

In the following algorithms, Π and Ψ are lists representing P and Q as lists of indices.

2.3. Decomposition algorithm

Although reverse-Ahnentafel indexing is again used in this section, the two occurrences of the RA function can be replaced with any other function that uniquely indexes the stripes and colonnades that might span a pivot block in your favorite data structure.

Notation. *Subscripts n,m,s, should be read as "north, middle, south"; and w,c,e as "west, central, east". No commas separate them.*

Using these enumerated types, instead of integers, for block indexing reads as easily as the compass, dodges debates about zero-based vs. one-based indexing, and obscures the artificial ordering among blocks.

Algorithm 2.1 (*Triangular decomposition*). to $\langle d, S, \Omega, \Pi, \Psi \rangle$, for $L + U'$ decomposition of non-singular A of order n. □

The decomposition results from GE using complete pivoting, with little said yet about how to select the pivots. Of course, pivoting determines the permutations that are reflected in Ω, Π, Ψ, lists of the orders of the pivot blocks, and of their (reverse-Ahnentafel) indices which determine the permutations P and Q. The matrix S ("sum") represents the permutation of $L + U'$: $S = P^{\mathrm{T}}(L + U')Q^{\mathrm{T}}$. The first result, d, is the determinant of A, modulo the sign of the permutations.

The HASKELL operators : and ++ are used for list CONSTRUCTION and concatenation. Complete, undulant-block pivoting is assumed, although no strategy for selecting pivot blocks is addressed here. No permutations are performed until Algorithm 2.2. Versions of these algorithms for quadtrees (cf. Section 3) have been programmed in SCHEME, C, and HASKELL, the international functional programming language [12].

A transformation from one sextuple $\langle A_l, d_l, S_l, \Omega_l, \Pi_l, \Psi_l \rangle$ to the next, $\langle A_{l+1}, d_{l+1}, S_{l+1}, \Omega_{l+1}, \Pi_{l+1}, \Psi_{l+1} \rangle$, is described below. An example is given in Fig. 4. Repeating it progressively zeroes out the first of these matrices, filling in the others. The initial input is $\langle A, 1, 0, [\,], [\,], [\,] \rangle$; the final result is $\langle 0, d, S, \Omega, \Pi, \Psi \rangle$.

l	A_l	d_l	S_l	Ω_l Π_l Ψ_l
0	$\begin{pmatrix} 3 & 1 & 2 & 1 \\ 1 & 1 & 1 & -1 \\ \boxed{4} & 3 & 4 & 1 \\ 4 & 2 & 3 & 1 \end{pmatrix}$	1	$\begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix}$	$\langle\,\rangle$ $\langle\,\rangle$ $\langle\,\rangle$
1	$\begin{pmatrix} -5/4 & \boxed{-1} & 1/4 & \\ 1/4 & \boxed{0} & \boxed{-5/4} & \\ & & & \\ -1 & -1 & 0 & \end{pmatrix}$	4	$\begin{pmatrix} -3/4 & & & \\ -1/4 & & & \\ 1/4 & 3 & 4 & 1 \\ -1 & & & \end{pmatrix}$	$\langle 1\rangle$ $\langle 5\rangle$ $\langle 4\rangle$
2	$\begin{pmatrix} & & & \\ & & & \\ & & & \\ \boxed{1/5} & & & \end{pmatrix}$	5	$\begin{pmatrix} -3/4 & -5/4 & -1 & -1/5 \\ -1/4 & 1/4 & 0 & -4/5 \\ 1/4 & 3 & 4 & 1 \\ -1 & & -1 & -1/5 \end{pmatrix}$	$\langle 1,2\rangle$ $\langle 5,2\rangle$ $\langle 4,3\rangle$
3	$\begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix}$	1	$\begin{pmatrix} -3/4 & -5/4 & -1 & -1/5 \\ -1/4 & 1/4 & 0 & -4/5 \\ 1/4 & 3 & 4 & 1 \\ -1 & 5 & -1 & -1/5 \end{pmatrix}$	$\langle 1,2,1\rangle$ $\langle 5,2,7\rangle$ $\langle 4,3,6\rangle$

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}; \qquad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} = Q.$$

$$L + U' = PS_3Q = \begin{pmatrix} 1/4 & 4 & 1 & 3 \\ \boxed{-3/4} & -1 & -1/5 & -5/4 \\ -1/4 & \boxed{0} & \boxed{-4/5} & 1/4 \\ -1 & -1 & \boxed{-1/5} & 5 \end{pmatrix}.$$

$$I - L = \begin{pmatrix} 1 & & & \\ \boxed{3/4} & 1 & 0 & \\ 1/4 & \boxed{0} & \boxed{1} & \\ 1 & 1 & 1/5 & 1 \end{pmatrix}; \qquad \begin{pmatrix} \boxed{4} & 4 & 1 & 3 \\ & \boxed{-1} & \boxed{1/4} & -5/4 \\ & 0 & -5/4 & 1/4 \\ & & & 1/5 \end{pmatrix} = U.$$

Fig. 4. Example of Algorithm 2.1: rational triangulation.

Let the block decomposition of A, isolating the $k \times k$ invertible pivot block, A_{mc}, be labeled

$$A = \begin{array}{c} \\ i \\ k \\ n-k-i \end{array} \begin{array}{c} \begin{array}{ccc} j & k & n-k-j \end{array} \\ \begin{pmatrix} A_{nw} & A_{nc} & A_{ne} \\ A_{mw} & A_{mc} & A_{me} \\ A_{sw} & A_{sc} & A_{se} \end{pmatrix} \end{array}$$

where A is $n \times n$ and A_{nw} is $i \times j$. (The local indices i, j, k are determined by the pivoting strategy.) The trivial case has $i = 0 = j$ and $k = n$.

Then decompose S_l similarly, to reflect elimination already done:

$$S_l = \begin{array}{c} \\ i \\ k \\ n-k-i \end{array} \begin{array}{c} \begin{array}{ccc} j & k & n-k-j \end{array} \\ \begin{pmatrix} S_{nw} & S_{nc} & S_{ne} \\ S_{mw} & 0 & S_{me} \\ S_{sw} & S_{sc} & S_{se} \end{pmatrix} \end{array}.$$

The center block is necessarily zero because A_{mc} is yet uneliminated; similarly, S_{nc} and A_{nc}, S_{mw} and A_{mw}, S_{me} and A_{me} S_{sc} and A_{sc} are disjoint submatrices, pairwise:

$$\Omega_{l+1} = \Omega_l + +[k]; \quad \Pi_{l+1} = \Pi_l + +[\text{RA}(i,k,n)]; \quad \Psi_{l+1} = \Psi_l + +[\text{RA}(j,k,n)].$$

Then compute

$$\langle S_{mc}, t\rangle = \langle A_{mc}^{-1}, \det A_{mc}\rangle, \qquad \begin{pmatrix} G_n \\ G_s \end{pmatrix} = \begin{pmatrix} A_{nc} \\ A_{sc} \end{pmatrix} (-S_{mc}).$$

The intermediate result, G should be viewed as a single colonnade (a Gauss vector [17, Section 3.2.1]) in the spanning space:

$$d_{l+1} = t \cdot d_l; \quad S_{l+1} = \begin{pmatrix} S_{nw} & S_{nc} \oplus G_n & S_{ne} \\ S_{mw} \oplus A_{mw} & S_{mc} & S_{me} \oplus A_{me} \\ S_{sw} & S_{sc} \oplus G_s & S_{se} \end{pmatrix}.$$

The direct sum, $\oplus$, can be used here because its terms are, respectively, disjoint.

$$A_{l+1} = \begin{pmatrix} A_{nw} & 0 & A_{ne} \\ 0 & 0 & 0 \\ A_{sw} & 0 & A_{se} \end{pmatrix} + \begin{pmatrix} G_n \\ 0 \\ G_s \end{pmatrix} (A_{mw} \quad 0 \quad A_{me}). \qquad \square$$

OpCount 2.3. *Algorithm* 2.1 *triangulates an* $n \times n$ *non-singular matrix within* $(n^3 - n^2)/3$ *scalar multiplications.*

As elimination proceeds on A_l, numeric information migrates from it to S_{l+1}, until the latter is empty. Fig. 4 presents an example decomposition algorithm on the matrix

$$\begin{pmatrix} 3 & 1 & 2 & 1 \\ 1 & 1 & 1 & -1 \\ 4 & 3 & 4 & 1 \\ 4 & 2 & 3 & 1 \end{pmatrix}$$

using strange pivot selections, indicated by boxes.

The permutations, P and Q, play an important role both in understanding the correctness and in the application of this algorithm; they are initially expressed as lists, Π, Ψ, and later as quadtree matrices, taking advantage of the representational efficiency for permutations from Table 1.

In the special case that $i = 0 = j$ at every level in the recurrence, no pivoting is done, and the algorithm might be called "undulant-block Gaussian elimination". In this case it is easy to see that $P = I$, $Q = I$, $S = L + U'$, and that these immediately form the $L + U'$ decomposition of A, by a quasidiagonal GE.

Theorem 2.2. *Let* $\langle d, S, \Omega, \Pi, \Psi\rangle$ *be the result from Algorithm* 2.1 *on non-singular input* A, *with* Π *and* Ψ *determining permutations* P *and* Q. *Then the* $L + U'$ *decomposition of* A *is* $\langle d(\det P)(\det Q), PSQ, \Omega, P, Q\rangle$.

Proof. If an oracle had provided P and Q before Algorithm 2.1 began, then we could have permuted A to PAQ, and the same relative pivoting on it would have followed the k blocks in its main quasidiagonal, D, to yield $(I - L)$ and U' without pivoting. Similarly, if that oracle had also decomposed the underlying vector space into the k subspaces corresponding to those blocks, which together form a new basis for the entire space, then the elimination above is isomorphic to the usual LU decomposition under the new k-dimension basis. At each step we would have observed $A_{nw} = A_{nc} = A_{ne} = A_{mw} = A_{sw} = 0$ and $S_{mc} = S_{me} = S_{sc} = S_{se} = 0$. Then, L is properly lower-triangular; U is upper diagonal (including D) and

$$PAQ = (I - L)U \quad \text{where } U' = U - D + D^{-1}$$

where D^{-1} is the aggregate of all A_{mc}^{-1} results, in situ.

Because *no* permutations occur during Algorithm 2.1, however, permuting A before elimination is equivalent to permuting its S_k result after all k pivot steps:

$$PS_kQ = L + U'. \qquad \square$$

Algorithm 2.2 ($L + U'$ *decomposition*). Use Algorithm 2.1 to triangulate A to $\langle d, S, \Omega, \Pi, \Psi \rangle$, translate Π and Ψ to P and Q and their parities, and apply the permutations *once* to yield $\langle d(\det P)(\det Q), PSQ, \Omega, P, Q \rangle$. $\square$

Numerical analysts will observe that it is usual *not* to invert the values in the quasidiagonal blocks; instead, they are reduced, recursively, to triangular $L + U$ so that the aggregate results are, themselves, perfectly triangular. The local permutations P and Q, which are here embedded in the local inverse of A_{mc}, must then be propagated through the pivot stripe and pivot colonnade, respectively. Those local permutations also become part of global results Π and Ψ but, that done, the computation of the Schur complement, A_{l+1} which is the *bulk* of the work in an elimination step, is the same, and the internal padding of Section 3.2 can be elided. The diagonal scalars in the triangular result are also not usually inverted (cf. $L + U$ in Fig. 3), because of asymmetries in floating-point representation. [3]

One can easily extend this algorithm to traditional, triangular LU decomposition, which is not pursued here for three reasons. First is a matter of art; this paper espouses undulant-block operations, and so extends them from GE through its trailer algorithms in the next section. Second, this paper sustains the algebraic kinship between floating-point and exact arithmetic by inverting undulant A_{mc} into S_{mc} under both types; the integer algorithm in Section 4 requires that the pivot block be inverted in order to retard

[3] A real number is approximated less accurately in memory, where it can be shared *among* processors, than in a processor's wider register, where it cannot be. Furthermore, a skew in the IEEE representation, that provides more representable magnitudes above one than reciprocals beneath it, and the usual, stable pivoting on scalars of larger magnitude suggest that scalar pivots be stored during triangulation even if their reciprocals must be recomputed following decomposition.

growth there of exact, intermediate results. Finally, immediate inversion abbreviates the indexing during decomposition and simplifies the associated permutations there; if they are expensive under your block representation then immediate inversion of A_{mc} becomes desirable.

A benefit of this emphasis on blocking is encapsulation of computation within the pivot stripe and colonnade, especially useful when they are wide. To clarify, this formulation holds invariant the blocks in the pivot stripe (A_{mw} and A_{me}) during the elimination step there, and those in the pivot column (L) later during the last step of inversion (cf. base cases of f,h in Algorithm 2.6.) With remote memory, caching, or paging [13], the pages of the pivot stripe or colonnade become "dirty" in just one of the two steps, and under parallelism the processes there similarly encapsulate more local results or none at all. Moreover, partial results are packaged for better sharing: computed and transmitted once to be used repeatedly by others.

There are four reasons to favor the block pivoting illustrated in Fig. 4 over traditional pivoting strategies (e.g. GE with partial pivoting).

1. The block structure of A and S in Fig. 4 follows the geography of the chosen pivot blocks, assuring a savings in intermediate storage under *any* matrix representation that favors block decomposition.
2. Algorithm 2.1 graciously admits pivots of undulant size, so that a larger pivot can be chosen to enhance parallelism when (a stable) one is available, but a smaller pivot can as easily be used if not. This parallelism occurs primarily in the computation of the Schur complement and also in computing the new column in S_{l+1}, where much local work is possible in each block. Both are done blockwise starting from A_{mc}.
3. It avoids repeated permutations, presumed here to be expensive on any block-oriented data structure and certainly expensive on quadtrees. Only one matrix permutation is performed, in Algorithm 2.2, to reorder *both* L and U', even though complete pivoting is allowed in Algorithm 2.1. Minimal permutations are required later in Algorithm 2.4, 2.5 or 2.6: just two vector permutations in the first case and only one row/column permutation in each of the latter two.
4. It admits both partial and complete pivoting naturally. Complete pivoting becomes more attractive in the context of undulant-block elimination because it is symmetric and so has a better chance of finding a larger pivot. It also becomes tractable when when there are fewer candidates of larger sizes; for example, quadtrees constrain 4×4 pivot candidates to align on every sixteenth matrix element; therefore, there are four times fewer possible pivot choices of that size than there are 2×2 candidates, and so the search is simpler.

Still ignoring the quadtree representation for a moment, let us visit three "trailer" algorithms to follow $L + U'$ decomposition to solve familiar problems. In all cases, when a matrix is decomposed: northwest, northeast, southwest, southeast; the division point is *between* blocks along the quasi-diagonal. That is, the northwest and southeast blocks each apply to a subspace determined by splitting Π and Ψ at the same place.

The algorithms in the following two subsections reduce to larger, localized multiplications when large diagonal blocks of L are zero – that is, when the associated quasidiagonal blocks in U', already inverted, are larger.

2.4. Determinant and solution algorithms

Algorithm 2.3 (*Determinant of A*). Use Algorithm 2.1 to triangulate A, revealing d, Π, and Ψ, and compute directly the parities of their corresponding permutation matrices, P and Q. Then, $\det A = d(\det P)(\det Q)$. □

Definition. A consistent partitioning of matrices M, X and vector $\boldsymbol{c}$ of the same order, and lists $\Omega, \Delta, \mathrm{T}$ (where Ω is a non-trivial list of orders summing to the order of the matrices), is labeling like the following:

$$\Omega = \langle \Omega_n, \Omega_s \rangle, \qquad \Delta = \langle \Delta_n, \Delta_s \rangle, \qquad \mathrm{T} = \langle \mathrm{T}_n, \mathrm{T}_s \rangle.$$

$$M = \begin{pmatrix} M_{nw} & M_{ne} \\ M_{sw} & M_{se} \end{pmatrix}, \qquad X = \begin{pmatrix} X_{nw} & X_{ne} \\ X_{sw} & X_{se} \end{pmatrix}, \qquad \boldsymbol{c} = \begin{pmatrix} \boldsymbol{c}_n \\ \boldsymbol{c}_s \end{pmatrix}.$$

The north and south halves of all the lists must be of the same order. The matrix partitions must have *nw* and *se* quadrants square – each of order equal the halves' of the vector, and to the sums of Ω_n and Ω_s, respectively.

Algorithm 2.4 (*Solving a linear system*). Solve $A\boldsymbol{x} = \boldsymbol{b}$ using the reformulation:

$$P^{-1}(I - L)UQ^{-1}\boldsymbol{x} = \boldsymbol{b}.$$

1. Compute the $L + U'$ decomposition of A using Algorithm 2.2.
2. (Forward substitution) Solve $(I - L)\boldsymbol{y} = P\boldsymbol{b} = \boldsymbol{c}$ using Ω.
   - If Ω is of length one then $L = 0$ and $\boldsymbol{y} = \boldsymbol{c}$.
   - Otherwise, consistently partition $\Omega = \langle \Omega_n, \Omega_s \rangle$ and

$$L + U' = \begin{pmatrix} L_{nw} + U'_{nw} & E \\ W & L_{se} + U'_{se} \end{pmatrix}, \qquad \boldsymbol{y} = \begin{pmatrix} \boldsymbol{y}_n \\ \boldsymbol{y}_s \end{pmatrix}, \qquad \boldsymbol{c} = \begin{pmatrix} \boldsymbol{c}_n \\ \boldsymbol{c}_s \end{pmatrix}.$$

     For efficiency, select a cleaving to make E and W of about the same size.
   - Recursively solve $(I - L_{nw})\boldsymbol{y}_n = \boldsymbol{c}_n$ using Ω_n.
   - Recursively solve $(I - L_{se})\boldsymbol{y}_s = \boldsymbol{c}_s + W\boldsymbol{y}_n$ using Ω_s.
3. (Backward substitution) Similarly, solve $U\boldsymbol{z} = \boldsymbol{y}$ using Ω recursively.
   - If Ω is of length one then $L + U' = U^{-1}$ and so $\boldsymbol{z} = (L + U')\boldsymbol{y}$.
   - Otherwise, partition Ω, $L + U'$, $\boldsymbol{y}$, and $\boldsymbol{c}$ as above.
   - Recursively solve $U_{se}\boldsymbol{z}_s = \boldsymbol{y}_s$.
   - Recursively solve $U_{nw}\boldsymbol{z}_n = \boldsymbol{y}_n - E\boldsymbol{z}_s$.
4. Permute $\boldsymbol{x} = Q\boldsymbol{z}$. □

Analogous HASKELL code appears as Fig. 5. In spite of functional style of this presentation, its data dependencies force the underlying algorithm to be serial. Back

```
type Decomp a = (a, Matrx a, Vectr Int, Matrx (), Matrx ())
solveLinear :: Decomp a -> Vectr a -> Vectr a
solveLinear (_, plusLU', omega, p, q)  b = (q #*| z) where
    y       = forwardSubst plusLU' (p #*| b) omega
    z       = backSubst    plusLU' y         omega
    forwardSubst :: Matrx a -> Vectr a -> Vectr Int -> Vectr a
    --forwardSubst plusLU' c       omega       = y   such that (I-L)y=c
    forwardSubst   _       c       (ScalarV _) = c                 -- I*c
    forwardSubst   _       ZeroV   _           = ZeroV
    forwardSubst (Mtx (plusLU'nw,_,w,plusLU'se)) (Vec c_n c_s)
                       (Vec omega_n omega_s) = (Vec y_n y_s) where
                        y_n = forwardSubst plusLU'n  c_n              omega_n
                        y_s = forwardSubst plusLU's (c_s + w #*| y_n) omega_s
    forwardSubst   _       _       _           =  error "Overconstrained"
    backSubst    :: Matrx a -> Vectr a -> Vectr Int -> Vectr a
    --backSubst plusLU' y      omega       = z   such that Uz=y
    backSubst   plusLU' y      (ScalarV _) = plusLU' #*| y     -- U'*y
    backSubst   _       ZeroV  _           = ZeroV
    backSubst   (Mtx (plusLU'nw,e,_,plusLU'se)) (Vec y_n y_s)
                      (Vec omega_n omega_s) = (Vec z_n z_s) where
                       z_n = backSubst plusLU'n (y_n - e #*| z_s) omega_s
                       z_s = backSubst plusLU's  y_s              omega_n
    ZeroM       #*| _           = ZeroV
    IdentM      #*| v           = v
    _           #*| ZeroV       = ZeroV
    (ScalarM x) #*| (ScalarV y) = ScalarV (x * y)
       --Except with infinitesimal floats:  case x*y of  0->0; z->ScalarV z
    Mtx (nw,ne,sw,se) #*| (Vec n s) = case (zipWith +
                                          (zipWith #*| (nw,ne) (n,s))
                                          (zipWith #*| (sw,se) (n,s))) of
                                               (ZeroV,ZeroV) -> ZeroV
                                               (n'   ,s'   ) -> Vec n' s'
```

Fig. 5. HASKELL code for Algorithm 2.4.

substitution necessarily proceeds southeast-to-northwest elong the quasidiagonal; parallelism is possible within matrix-vector multiplications that coordinate on its decomposition. As in Algorithm 2.1 larger quasi-diagonal blocks will improve parallelism.

OpCount 2.4. *Algorithm* 2.4 *solves a linear system of order n within* $(n^3 + 2n^2)/3$ *scalar multiplications* (*exclusive of permutations*).

The coefficient on n^2, which is the difference from OpCount 2.3, shrinks as larger non-scalar blocks were eliminated in Algorithm 2.2; still, the first term dominates.

2.5. *Inversion*

The sequentiality implicit in Algorithm 2.4 contrasts with ample parallelism in the following two algorithms. Algorithm 2.5, which multiplies triangular inverses, is familiar in its iterative formulation; however, Algorithm 2.6 is better.

Algorithm 2.5 (*Matrix inversion*). Invert A to $\langle A^{-1}, \det A \rangle$.

1. Compute the $L + U'$ decomposition of A using Algorithm 2.2, yielding $\langle d, L + U', \Omega, P, Q\rangle$.
2. Compute $\hat{L} = (I - L)^{-1}$ and $\hat{U} = U^{-1}$ recursively.
   - If Ω is of length one, then $L = 0$, $\hat{L} = I$, and $\hat{U} = U' = U^{-1}$.
   - Otherwise, partition $L + U'$ and Ω as in Algorithm 2.4.
   - Recursively (even simultaneously) compute

$$\hat{U}_{nw} = U_{nw}^{-1}, \qquad \hat{U}_{se} = U_{se}^{-1},$$

$$\hat{L}_{nw} = (I - L_{nw})^{-1}, \qquad \hat{L}_{se} = (I - L_{se})^{-1}.$$

   - Then

$$\hat{U} = \begin{pmatrix} \hat{U}_{nw} & -\hat{U}_{nw} E \hat{U}_{se} \\ 0 & \hat{U}_{se} \end{pmatrix}, \qquad \hat{L} = \begin{pmatrix} \hat{L}_{nw} & 0 \\ \hat{L}_{se} W \hat{L}_{nw} & \hat{L}_{se} \end{pmatrix}.$$

3. $\langle A^{-1}, \det A\rangle = \langle Q(\hat{U}\hat{L})P, d\rangle$. □

OpCount 2.5. *Algorithm* 2.5 *inverts an* $n \times n$ *matrix within* $n^3 - 2n^2/3$ *multiplications (exclusive of permutations).*

Plenty of parallelism is available because the northwest and southeast inverses, as well as manipulations of L and U, are independent of one another. The breakdown of multiplication counts, ignoring the permutations, is $n^3/3$ (OpCount 2.2) at Step 1, $n^3/6 + O(n^2)$ for each triangular inversion at Step 2, and $n^3/3$ to multiply the triangular matrices (OpCount 1.3) at Step 3. However, the same number of multiplications yield the result with far more locality and parallelism in Algorithm 2.6, which absorbs Step 3 into Steps 1 and 2.

Algorithm 2.6 (*Matrix inversion*). Invert A to $\langle A^{-1}, \det A\rangle$. A block-recursive definition of the function f follows these two steps:
1. Compute the $L + U'$ decomposition of A using Algorithm 2.2.
2. Compute $A^{-1} = Q[f(L + U', \Omega)]P$.

The three functions, f, g, h are defined mutually and computed recursively:
- Define $f : (L + U', \Omega) \mapsto M$, where M is the same size as $L + U'$, as follows:
  - If Ω is of length one, then $M = L + U'$.
  - Otherwise, consistently partition

$$\Omega = \langle \Omega_n, \Omega_s\rangle,$$

$$L + U' = \begin{pmatrix} L_{nw} + U'_{nw} & E \\ W & L_{se} + U'_{se} \end{pmatrix}.$$

Compute $K = f(L_{se} + U'_{se}, \Omega_s)$ and $F = f(L_{nw} + U'_{nw}, \Omega_n)$.
Compute $B = g(L_{nw} + U'_{nw}, E, \Omega_n)$ and $C = h(L_{nw} + U'_{nw}, W, \Omega_n)$.

Compute $H = BK$ and $J = KC$; then $G = F + BJ$ and

$$M = \begin{pmatrix} G & H \\ J & K \end{pmatrix}.$$

- Define $g : (L + U', X, \Omega) \mapsto M$, with M of the same size as X and compatible with $L + U'$ in its number of rows, as follows:
  - If Ω is of length one, then $M = -(L + U')X$.
  - Otherwise, consistently partition

$$\Omega = \langle \Omega_n, \Omega_s \rangle,$$

$$L + U' = \begin{pmatrix} L_{nw} + U'_{nw} & E \\ W & L_{se} + U'_{se} \end{pmatrix}, \qquad X = \begin{pmatrix} X_{nw} & X_{ne} \\ X_{sw} & X_{se} \end{pmatrix}.$$

Compute $J = g(L_{se} + U'_{se}, X_{sw}, \Omega_s)$ and $K = g(L_{se} + U'_{se}, X_{se}, \Omega_s)$.
Compute $G = g(L_{nw} + U'_{nw}, X_{nw} + EJ, \Omega_n)$ and
$H = g(L_{nw} + U'_{nw}, X_{ne} + EK, \Omega_n)$.
Finally, assemble

$$M = \begin{pmatrix} G & H \\ J & K \end{pmatrix}.$$

- Define $h : (L + U', X, \Omega) \mapsto M$, with M of the same size as X and compatible with $L + U'$ in its number of columns, as follows:
  - If Ω is of length one, then $M = X$.
  - Otherwise, consistently partition

$$\Omega = \langle \Omega_n, \Omega_s \rangle,$$

$$L + U' = \begin{pmatrix} L_{nw} + U'_{nw} & E \\ W & L_{se} + U'_{se} \end{pmatrix}, \qquad X = \begin{pmatrix} X_{nw} & X_{ne} \\ X_{sw} & X_{se} \end{pmatrix}.$$

Compute $H = h(L_{se} + U'_{se}, X_{ne}, \Omega_s)$ and $K = h(L_{se} + U'_{se}, X_{se}, \Omega_s)$.
Compute $G = h(L_{nw} + U'_{nw}, X_{nw} + HW, \Omega_n)$ and
$J = h(L_{nw} + U'_{nw}, X_{sw} + KW, \Omega_n)$.
Finally, assemble M as in the definition of g, above. □

While the northwest and southeast quadrants in these consistent decompositions must be square, nothing yet requires them to have the same order. At this point their compatible northeast and southwest quadrants could be rectangular; they only appear as factors in the recurrences above.

Theorem 2.3.

$$f(L + U', \Omega) = U^{-1}(I - L)^{-1},$$

$$g(L + U', X, \Omega) = -U^{-1}X,$$

$$h(L + U', X, \Omega) = X(I - L)^{-1}.$$

Proof. By mutual, course-of-values induction on the length of Ω.

Bases: If the length of Ω is one, then $L=0$ and

$$f(L+U',\Omega)=U'=U^{-1}=U^{-1}(I-L)^{-1},$$

$$g(L+U',X,\Omega)=-U'X=-U^{-1}X,$$

$$h(L+U',X,\Omega)=X=X(I-L)^{-1}.$$

Induction step: Assume that the theorem holds for all decompositions whose associated Ω sequences of sizes are properly less than that of this one, and prove the theorem for this decomposition. Decompose $\Omega=\langle\Omega_n,\Omega_s\rangle$, with Ω_n,Ω_s both non-empty; decompose consistently $L+U'$ and

$$X=\begin{pmatrix}X_{nw} & X_{ne}\\ X_{sw} & X_{se}\end{pmatrix},$$

as required above (for f,g,h in turn). Then

$$\begin{aligned}&(I-L)U[f(L+U',\Omega)]\\ &=\begin{pmatrix}I-L_{nw} & 0\\ -W & I-L_{se}\end{pmatrix}\begin{pmatrix}U_{nw} & E\\ 0 & U_{se}\end{pmatrix}\\ &\quad\times\begin{pmatrix}f(L_{nw}+U'_{nw},\Omega_n)+BJ & [B=g(L_{nw}+U'_{nw},E,\Omega_n)]K\\ J=K[h(L_{nw}+U'_{nw},W,\Omega_n)] & K=f(L_{se}+U'_{se},\Omega_s)\end{pmatrix}.\end{aligned}$$

Using the inductive hypothesis:

$$\begin{aligned}&=\begin{pmatrix}(I-L_{nw})U_{nw} & (I-L_{nw})E\\ -WU_{nw} & -WE+(I-L_{se})U_{se}\end{pmatrix}\\ &\quad\times\begin{pmatrix}U_{nw}^{-1}(I-L_{nw})^{-1}-U_{nw}^{-1}EU_{se}^{-1}(I-L_{se})^{-1}W(I-L_{nw})^{-1} & -U_{nw}^{-1}EU_{se}^{-1}(I-L_{se})^{-1}\\ U_{se}^{-1}(I-L_{se})^{-1}W(I-L_{nw})^{-1} & U_{se}^{-1}(I-L_{se})^{-1}\end{pmatrix}\\ &=I,\end{aligned}$$

$$\begin{aligned}&U[g(L+U',X,\Omega)]\\ &=\begin{pmatrix}U_{nw} & E\\ 0 & U_{se}\end{pmatrix}\begin{pmatrix}g(L_{nw}+U'_{nw},X_{nw}+EJ,\Omega_n) & g(L_{nw}+U'_{nw},X_{ne}+EK,\Omega_n)\\ J=g(L_{se}+U'_{se},X_{sw},\Omega_s) & K=g(L_{se}+U'_{se},X_{se},\Omega_s)\end{pmatrix}.\end{aligned}$$

Using the inductive hypothesis,

$$\begin{aligned}&=\begin{pmatrix}U_{nw} & E\\ 0 & U_{se}\end{pmatrix}\begin{pmatrix}-U_{nw}^{-1}(X_{nw}-EU_{se}^{-1}X_{sw}) & -U_{nw}^{-1}(X_{ne}-EU_{se}^{-1}X_{se})\\ -U_{se}^{-1}X_{sw} & -U_{se}^{-1}X_{se}\end{pmatrix}\\ &=-X.\end{aligned}$$

$$
\begin{aligned}
&[h(L+U',X,\Omega)](I-L)\\
&=\begin{pmatrix} h(L_{nw}+U'_{nw},X_{nw}+HW,\Omega_n) & H=h(L_{se}+U'_{se},X_{ne},\Omega_s)\\ h(L_{nw}+U'_{nw},X_{sw}+KW,\Omega_n) & K=h(L_{se}+U'_{se},X_{se},\Omega_s)\end{pmatrix}\\
&\quad\times\begin{pmatrix} I-L_{nw} & 0\\ -W & I-L_{se}\end{pmatrix}.
\end{aligned}
$$

Using the inductive hypothesis,

$$
\begin{aligned}
&=\begin{pmatrix} (X_{nw}+X_{ne}(I-L_{se})^{-1}W)(I-L_{nw})^{-1} & X_{ne}(I-L_{se})^{-1}\\ (X_{sw}+X_{se}(I-L_{se})^{-1}W)(I-L_{nw})^{-1} & X_{se}(I-L_{se})^{-1}\end{pmatrix}\\
&\quad\times\begin{pmatrix} I-L_{nw} & 0\\ -W & I-L_{se}\end{pmatrix},\\
&=X. \qquad \square
\end{aligned}
$$

Corollary 2.2. *Algorithm* 2.6 *computes the inverse of a matrix.*

Corollary 2.3. *The quadrant,* $G=F+BJ$, *in the definition of* f *in Algorithm* 2.6 *can also be computed as* $G=F+HC$.

Proof. Using the notation there, $F+BJ=F+BKC=F+HC$.

Corollary 2.4. *When the second argument to either of* g,h *is* 0, *their results are annihilated to* 0.

This corollary leads to the code sketched in Fig. 6. It also implies that these algorithms will be very fast on sparse matrices, because the only operations are direct recursions, multiplications, and additions – all of which collapse on zero arguments.

OpCount 2.6. *If the* $n\times n$ *matrix* $L+U'$ *and* X, *result from scalar eliminations, then the function* g *can be applied within aggregate* $n^3/2$ *multiplications.*

OpCount 2.7. *If the* $n\times n$ *matrix* $L+U'$ *and* X, *result from scalar eliminations, then the function* h *can be applied within aggregate* $n^3/2-n^2$ *multiplications.*

Validation of this count is simplest when the X matrix is square although it need not be. The term, "aggregate", covers the case that it is compatible but not square; then the multiplication count accrues over many applications during inversion of a square matrix, to the stated total overall.

OpCount 2.8. *The function* f *can be applied to an* $n\times n$ *triangulation within* $(2n^3-n^2)/3$ *multiplications.*

```
g :: Num a => (Matrx a) -> (Matrx a) -> (Vectr Int) -> (Matrx a)
g _          ZeroM  _                    = ZeroM
g l_plus_u' xx      (ScalarV _)          = negate (l_plus_u' * xx)
g (Mtx (l_plus_u'_nw,     ee,      _, l_plus_u'_se))
  (Mtx (        xx_nw, xx_ne, xx_sw,         xx_se))
  (Vec omega_n omega_s)           = normalize (gg, hh, jj, kk) where
      jj = g l_plus_u'_se xx_sw                omega_s
      kk = g l_plus_u'_se xx_se                omega_s
      gg = g l_plus_u'_nw (xx_nw + ee * jj) omega_n
      hh = g l_plus_u'_nw (xx_ne + ee * kk) omega_n

h :: Num a => (Matrx a) -> (Matrx a) -> (Vectr Int) -> (Matrx a)
h _          ZeroM  _                    = ZeroM
h l_plus_u' _       (ScalarV _)          = l_plus_u'
h (Mtx (l_plus_u'_nw,     _,     ww, l_plus_u'_se))
  (Mtx (        xx_nw, xx_ne, xx_sw,         xx_se))
  (Vec omega_n omega_s)           = normalize (gg, hh, jj, kk) where
    hh = h l_plus_u'_se xx_ne                omega_s
    kk = h l_plus_u'_se xx_se                omega_s
    gg = h l_plus_u'_nw (xx_nw + hh * ww) omega_n
    jj = h l_plus_u'_nw (xx_sw + kK * ww) omega_n
```

Fig. 6. HASKELL sketch of g and h.

OpCount 2.9. *Algorithm* 2.6 *inverts an* $n \times n$ *matrix within* $n^3 - 2n^2/3$ *multiplications, exclusive of permutations.*

OpCounts 2.5 and 2.9 would suggest that Algorithm 2.5 and 2.6 are of equivalent complexity. On a multiprocessor, however, the former algorithm does not break down into independent processes as nicely. That is, after its decomposition (only 1/3 of the measure) Algorithm 2.6 only invokes f, which decomposes at each level into successively *nested* invocations of f, g, h without returning to 'top level.' In contrast, Algorithm 2.5 computes $(I - L)^{-1}$ and U^{-1}, and *then* multiplies them; it returns to higher levels to schedule further traversal of matrices of order n and smaller, at a cost in locality. In effect, Algorithm 2.6 distributes the final multiplication across the two triangular inversions, with additional locality that allows superior parallelism. Since most of the computation in f accrues on the southeast and northwest quadrants, moreover, the access patterns there remain local even with a row-major or column-major array representation.

Algorithm 2.1 is merely a generalization of generic LU decomposition. If it were restricted, so that o_l were fixed for all l, then it would implement conventional (non-undulant) block pivoting; if $o_l = 1$ everywhere, it implements scalar pivoting. If it is further restricted, so $A_{nw} = A_{nc} = A_{ne} = 0$ at all steps, then $P = I$ and it implements partial pivoting. If, furthermore, $A_{mw} = A_{sw} = 0$ then $Q = I$, pivoting disappears, and Algorithm 2.1 implements traditional Gaussian elimination. These restrictions may be imposed a differents order to yield other familiar cases.

The generalization was necessary in order to discuss block pivoting with undulant (varying) block size: o_l varies. In order to take best advantage of efficient BLAS

Level 2 and 3 (block) operations, we would like o_l to be large at some steps even though constraints on pivoting may force it to be smaller at others. If the choice of pivot were unrestricted, however, then pivot selection can become more difficult than the underlying problem.

A convenient example is P in Fig. 4; it has only one contiguous 2×2 block (of nine possible) that is non-singular and, therefore, could be eliminated if we were inverting P, itself. A search for it that also considers all eight alternatives or (worse) non-contiguous blocks, however, explodes on matrices that are any larger. The suggested solution is to return to the data structure, searching only among *representable* blocks for a pivot; that is, search among subtrees of the quadtree representation.

3. Decomposition of quadtree matrices

> Make no mistake about it: Computers process numbers – not symbols. We measure our understanding (and control) by the extent to which we can arithmetize an activity. [18, ¶65]

The algorithms of Section 2 have a common philosophy of decomposing matrices into blocks. This section marries those algorithms to the quadtree representation of Section 1. That is, the decomposition is now to follow the boundaries imposed by the quadtree representation: all blocks will be represented as subtrees and have reverse-Ahnentafel indices. Specifically, the A_{mc} blocks that are eliminated in Algorithm 2.1 *must* coincide with subtrees. This constraint limits pivoting to only a relatively few candidates.

3.1. Recursive decomposition

Fig. 7 helps guide our reformulation of Algorithm 2.1 to fit the quadtree data structure. The top figures sketch the core of the elimination of Block A_{mc} at the $(l+1)$th elimination step. They are to be fitted to the quadrant decomposition, so that A_{mc}, itself, never crosses a quadrant boundary. This implies that the stripe and colonnade coordinating on it casts predictable patterns across the four quadrants, illustrated in the four smaller blocks at the bottom of Fig. 7.

Let us identify four blocks that occur in those patterns. A_{mc} lands in one, which will be called the PIV block. The one oriented diagonally from PIV is called OFF; no elimination occurs there during this step, and its uneliminated elements are fodder for the Schur complement. It the property that, if decomposed, it yields four OFF quadrants. The last two are *adjacent* to the PIV block. The one oriented horizontally, either east or west, from PIV is called ROW and contains an extension of the stripe to be eliminated; it has the property that, if decomposed, it cleaves into two OFF and two ROW quadrants. The last oriented vertically, either north or south, from PIV is called COL and contains an extension of the colonnade to be eliminated; it has the property that, if decomposed further, it cleaves into two OFF and two COL quadrants.

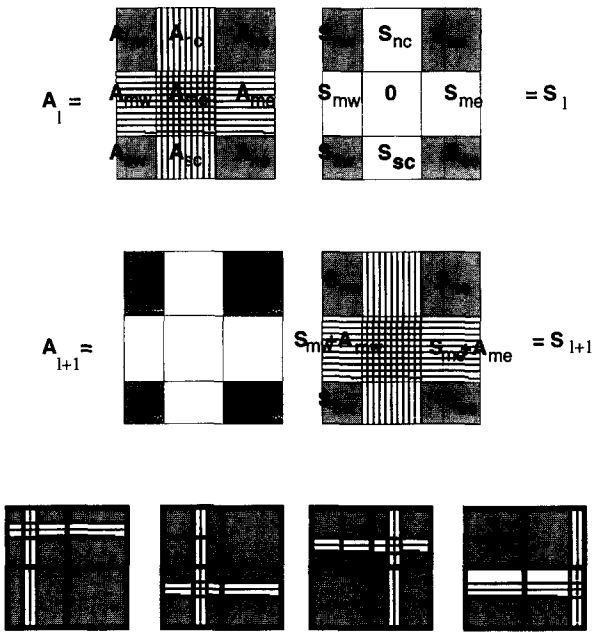

Fig. 7. Triangular-decomposition step, and quadtree matrices.

This characterization applies regardless of whether PIV occurs northwest, southwest, northeast, or southeast as illustrated left-to-right at the bottom of Fig. 7. The northwest quadrants of each of those four sketches illustrate the recursive decompositions of each of PIV, ROW, COL, and OFF, respectively. The PIV block, itself, splits into four quadrants: one of each of the four kinds; that is, A_l itself is a PIV block, rooting the recursive programs below.

Definition. A tree is decorated when it has additional atomic information stored at internal nodes.

A later algorithm requires that the quadtree structure for the various A_l be decorated, but that those for S_l, $L + U'$, and permutation matrices remain undecorated. Both decorated and undecorated quadtrees have the elements of the represented matrix stored at their leaves.

Since HASKELL is a strongly typed language, it requires many types of products, especially for heterogeneous matrix-multiplication because its heterogeneous operators cannot be overloaded. Therefore, the following conventions are followed here; see Figs. 4 and 7.

Notation. In this HASKELL code, operators + and * are sometimes festooned with characters like #, @, | and . on the left and right to indicate the type of its two operands.

```
infixl [6] #+%
infixl [7] #*@ , @*#
type DecorQuadrants a = (DecorMatrx a, DecorMatrx a, DecorMatrx a, DecorMatrx a)
data DecorMatrx a =  ZeroD | ScalarD a | MtxD  (Decoration a) (DecorQuadrants a)
instance (Num a) => Num (DecorMatrx a)
decorate:: (DecorQuadrants a) -> (DecorMatrx a)
decorate (ZeroD,ZeroD,ZeroD,ZeroD) = ZeroD
decorate quads                     = MtxD (decor quads) quads
decor::  (DecorQuadrants a) -> (Decoration a)
```

Fig. 8. Declarations for decorated matrices.

The first two suggest a matrix – respectively undecorated and decorated; the third suggests a vector and the forth suggests a scalar. Thus, `#*@` is matrix product for an undecorated matrix times a decorated matrix; `@*#` is matrix product for a decorated matrix times an undecorated matrix; `#+@` is matrix sum of an undecorated matrix and a decorated matrix; `#*|` is an undecorated-matrix–vector product, etc. (see Fig. 8).

Homogeneous operators, however, can be overloaded using HASKELL's preferred `class/instance` protocol; thus, plain old `*` will be used where `#*#` might have been expected (cf. Fig. 2.)

3.1.1. Shur complement from matrix elimination

Fig. 9 presents typical HASKELL code for computing the Shur complement of the OFF quadrant. As illustrated in the northwest of the rightmost matrix at the bottom of Fig. 7, an OFF quadrant decomposes into four OFF quadrants. This function, named `off`, takes four arguments and yields just a single result. Its arguments correspond to a subblock from $A_{l,nw}$, the projection of it on the Gauss vector (or colonnade): $-A_{l,nc}A_{l,mc}^{-1}$; and its projection from the pivot row (respectively, stripe): $A_{l,mw}$ together with i – the reverse-Ahnentafel index to that stripe. The index also identifies the base of the recurrence, which can be a non-trivial block. The code for `off` follows a straightforward recurrence: after handling cases that render the Shur complement an identity (when either pivot row or pivot column is zero), it decomposes, either to a negative outer-product, to a single matrix multiply-add operation (a *mop*, analogous to a *flop*), or two four parallel and mutually independent process. This latter case occurs most often, and its obvious parallelism here is a strength of this formulation for GE and of this algorithm.

3.1.2. Elimination from adjacent quadrants

The next-to-least complicated code handles the ROW block (Fig. 10), which is either the same size as the eliminated block, or decomposes into two ROW and two OFF blocks. This is illustrated in the second matrix from the right at the bottom of Fig. 7. The second two have arguments that are results from the first two; lazy evaluation provides a neat way to overlap all four but, in general, this dependency suggests that computation of the second pair follow (in time) that of the first pair. However, each pair can be performed as parallel processes.

```
-- off          A                gaussS              pivRowA          i
--      yields  A'
off:: (Num a) => DecorMatrx a -> Vectr (Matrx a) -> DecorMatrx a -> Int
        -> DecorMatrx a
off   a             _       ZeroD _ =  a
off   a             ZeroV   _     _ =  a
off   ZeroD         gauss   pivR  i =  (outerProduct i gauss pivR) where
        outerProduct :: Int -> Vec a -> DecorMatrx a -> DecorMatrx a
        outerProduct    _       ZeroV          _       =  ZeroD
        outerProduct    _       _              ZeroD   =  ZeroD
        outerProduct    1     (ScalarV ca')    b       =  ca' #*@ b
        outerProduct    i     (Vec gauss_n gauss_s)
                                          (MtxD _ (nw, ne, sw, se)) =
            decorate (zipWith3 outerProduct iResidues pCols pRows) where
                    iResidues   = (half,half,half,half) where  half = div i 2
                    pCols       = (gauss_n, gauss_n, gauss_s, gauss_s)
                    pRows |even i     = (nw,ne,nw,ne)
                          |otherwise  = (sw,se,sw,se)
off   d             (ScalarV ca') b  1 =  d + ca' #*@ b
off (MtxD _ aQuads) (Vec gauss_n gauss_s)
                          (MtxD _ (r_nw, r_ne, r_sw, r_se)) i =
        decorate (zipWith4 off aQuads pCols pRows iResidues) where
                    iResidues   = (half,half,half,half) where  half = div i 2
                    pCols       = (gauss_n, gauss_n, gauss_s, gauss_s)
                    pRows |even i     = (r_nw, r_ne, r_nw, r_ne)
                          |otherwise  = (r_sw, r_se, r_sw, r_se)
(#*@):: Matrx a -> DecorMatrx a -> DecorMatrx a
ZeroM           #*@  _             =  ZeroD
_               #*@  ZeroD         =  ZeroD
ScalarM x       #*@  ScalarD y     =  ScalarD (x*y)
   --Except with infinitesimal floats:  case x*y of  0->0; z->ScalarD z
Mtx x           #*@  MtxD _ y      =  decorate (zipWith (+)
                        (zipWith (#*@) (colExchange x)(offDiagSqsh y))
                        (zipWith (#*@)       x        (prmDiagSqsh y)) )
```

Fig. 9. HASKELL code for OFF quadrant's Shur complement.

Row's arguments are the portions of A_l and S_l that land in this quadrant; gauss which is the binary-vector projection of

$$\begin{pmatrix} G_n \\ 0 \\ G_s \end{pmatrix}$$

on this quadrant; and i to identify the pivot row (stripe) within it. The parity of i determines whether the pivot stripe traverses the north or the south half of this quadrant, orienting the recursions accordingly.

Col, on the other hand (Fig. 11), has two more parameters and one more result. It performs elimination on the COL block, which is illustrated in the second block from the left at the bottom of Fig. 7; it decomposes into two COL blocks and two OFF blocks. Again, parity – now of j – determines whether the pivot colonnade traverses the west or the east half of this quadrant, orienting the recursions accordingly.

Again, the second two recurrences have arguments that are results from the first two and laziness provides a way to overlap all four subcomputations; typically, however, the

```
-- row    A          S           gaussS              i yields  (A'   , S')
row :: (Num a) =>
    DecorMatrx a -> Matrx a -> Vectr (Matrx a) -> Int -> (DecorMatrx a, Matrx a)
row ZeroD s _      _ =  (ZeroD, s)
row a      _ _     1 =  (ZeroD, s #+% a) where
        ZeroM     #+% a          = undecorate a  where
            undecorate ZeroD           = ZeroM
            undecorate ScalarD x       = ScalarM x
            undecorate (MtxD _ quads) = Mtx (map undecorate quads)
        s         #+% ZeroD         = s
        ScalarM x #+% ScalarD y = case x+y of z |z==fromInteger 0  -> ZeroM
                                                |otherwise         -> ScalarM z
        MtxS x    #+% MtxD _  y = normalize (zipWith (#+%) x y)
row a     s ZeroV i =  rowExtract a s i where
        rowExtract ZeroD                          s      _ = (ZeroD, s)
        rowExtract (ScalarD x)                    ZeroM 1 = (ZeroD, ScalarM x)
        rowExtract (Mtx _ (a_nw, a_ne, a_sw, a_se)) s     i =
           let iResidue = div i 2
               (s_nw, s_ne, s_sw, s_se) |s==ZeroM   = (ZeroM,ZeroM,ZeroM,ZeroM)
                                        |otherwise  = quads where Mtx quads = s
           in case (even i) of
               True->  (decorate  (a'_nw, a'_ne, a_sw, a_se),
                        normalize (s'_nw, s'_ne, s_sw, s_se)) where
                   (a'_nw, s'_nw) = rowExtract a_nw s_nw iResidue
                   (a'_ne, s'_ne) = rowExtract a_ne s_ne iResidue
               False-> (decorate  (a_nw, a_ne, a'_sw, a'_se),
                        normalize (s_nw, s_ne, s'_sw, s'_se)) where
                   (a'_sw, s'_sw) = rowExtract a_sw s_sw iResidue
                   (a'_se, s'_se) = rowExtract a_se s_se iResidue
row (DMtx _ (a_nw, a_ne, a_sw, a_se)) s (Vec gauss_n gauss_s) i =
    let iResidue = div i 2
        (s_nw, s_ne, s_sw, s_se) |s==ZeroM   = (ZeroM,ZeroM,ZeroM,ZeroM)
                                 |otherwise  = quads where (Mtx quads) = s
    in case (even i) of
        True->  (decorate  (a'_nw, a'_ne, a'_sw, a'_se),
                 normalize (s'_nw, s'_ne, s_sw,  s_se )) where
                   (a'_nw, s'_nw) = row a_nw s_nw gauss_n iResidue
                   (a'_ne, s'_ne) = row a_ne s_ne gauss_n iResidue
                   a'_sw          = off a_sw gauss_s a_nw iResidue
                   a'_se          = off a_se gauss_s a_ne iResidue
        False-> (decorate  (a'_nw, a'_ne, a'_sw, a'_se),
                 normalize (s_nw,  s_ne,  s'_sw, s'_se)) where
                   (a'_sw, s'_sw) = row a_sw s_sw gauss_s iResidue
                   (a'_se, s'_se) = row a_se s_se gauss_s iResidue
                   a'_nw          = off a_nw gauss_n a_sw iResidue
                   a'_ne          = off a_ne gauss_n a_se iResidue
```

Fig. 10. HASKELL code for elimination from the ROW quadrant.

first two (parallel) processes will be completed before the second two begin. The extra result is the aforementioned Gauss vector (pivot colonnade), which is readily available neither in A_l because it is a product of blocks, nor in S_{l+1} because it is embedded in a direct sum there. Rather than extracting it, we construct a binary vector explicitly (and then reuse its space later) because it decomposes nicely with the GE recursion. The new parameters are the reverse-Ahnentafel column index j, and the negation of the inverted pivot block, $-A_{l,mc}^{-1}$, a right factor to all products that build the Gauss vector.

The cases are straightforward, except to note that the pivot row in A_l projecting on this quadrant cannot be entirely zero because it contains the pivot block, itself

```
-- col    A          S          pivRow          i      j      V
--    yields  (A'  ,     S',     gauss)
col :: (Num a) =>
    DecorMatrx a -> Matrx a -> DecorMatrx a -> Int -> Int -> Matrx a
     -> (DecorMatrx a, Matrx a, Vectr (Matrx a))
col ZeroD s _ _ _ _ =  (ZeroD, s. ZeroV)
col a s r 1 1 minus_v =  (ZeroD,  ca', ScalarV ca') where
    ca'                      = a @*# minus_v
    ZeroD     @*# _          = ZeroM
    _         @*# ZeroM      = ZeroM
    ScalarD x @*# ScalarM y = ScalarM (x*y)
       --Except with infinitesimal floats:  case x*y of  0->0; z->ScalarM z
    MtxD _ x  @*# Mtx y      = normalize (zipWith (+)
                             (zipWith (@*#) (colExchange x)(offDiagSqsh y))
                             (zipWith (@*#)       x         (prmDiagSqsh y)) )
col (DMtx _ aQuads) s (MtxD _ (r_nw, r_ne, r_sw, r_se)) i j minus_v =
    let iResidue = div i 2
        jResidue = div j 2
        (s_nw, s_ne, s_sw, s_se) |s==ZeroM   = (ZeroM,ZeroM,ZeroM,ZeroM)
                                 |otherwise  = quads where (Mtx quads) = s
        (row_w, row_e)           |even i     = (r_nw, r_ne)
                                 |otherwise  = (r_sw, r_se)
    in case (even j) of
        True->  (decorate  (a'_nw, a'_ne, a'_sw, a'_se),
                 normalize (s'_nw, s_ne , s'_sw, s_se ),
                 case (gauss_n, gauss_s) of (ZeroV,ZeroV) -> ZeroV
                                            (n    ,s    ) -> Vec n s )
                where
                   (a'_nw, s'_nw, gauss_n)  =
                                col a_nw s_nw row_w iResidue jResidue minus_v
                   (a'_sw, s'_sw, gauss_s)  =
                                col a_sw s_sw row_w iResidue jResidue minus_v
                   a'_ne                    = off a_ne gauss_n row_e iResidue
                   a'_se                    = off a_se gauss_s row_e iResidue
        False-> (decorate  (a'_nw, a'_ne, a'_sw, a'_se),
                 normalize (s_nw,  s'_ne,  s_sw, s'_se),
                 case (gauss_n, gauss_s) of (ZeroV,ZeroV) -> ZeroV
                                            (n    ,s    ) -> Vec n s )
                where
                   (a'_ne, s'_ne, gauss_n)  =
                                col a_ne s_ne row_e iResidue jResidue minus_v
                   (a'_se, s'_se, gauss_s)  =
                                col a_se s_se row_e iResidue jResidue minus_v
                   a'_nw                    = off a_nw gauss_n row_w iResidue
                   a'_sw                    = off a_sw gauss_s row_w iResidue
```

Fig. 11. HASKELL code for elimination from the COL quadrant.

nonsingular. The base case, when $i = 1 = j$, identifies a block that spans the pivot colonnade.

3.1.3. *Elimination from the* PIV *quadrant*

Finally, we consider to the code for `piv` in Fig. 13. It seems fairly complicated at first but, except for the basis of its recursion, the complexities of its computation have already been described. The only remaining complexity is its type: why so few parameters and so many results?

To understand PIV, we anticipate Section 3.3 on decorating matrices for pivoting. Descriptively, each interior node in a decorated matrix contains a *signpost* (Fig. 12)

```
type Decoration a = (Signpost, More)
type Signpost = (Boolean, Boolean, TinyInt)
```

`TinyInt` is intended to be a 6-bit integer; in PASCAL we would write its type as `(0..63)`. Thus, a `Signpost` occupies only a byte. This limits the order of a matrix to 2^{63}, maybe a bit more if "Scalars" (leaves) were nontrivial: a modest constraint.
`More` is just a placeholder for additional decorations.

Fig. 12. Declarations for signposts and decorations.

```
--piv: depth -> A -> S -> (A', V, S', gauss, i, j, det)
piv:: (Fractional a) => Int -> DecorMatrx  a -> Matrx a
    -> (DecorMatrx  a, Matrx a, Matrx a, Vectr (Matrx a), Int, Int, a)

piv 0 a ZeroM = (ZeroD, negate v, v, ZeroV, 1, 1, det) where
                (v, det) = invert a
piv (depth+1) (DMtx ((north, west, _): _) aQuads)  s =
    let ident x                     = x
        vecIdentity north south     = Vec north south
        vecExchange north south     = Vec south north
        rowExchange  (nw,ne,sw,se) = (sw,se,nw,ne)
        diagExchange (nw,ne,sw,se) = (se,sw,ne,nw)
        leftSon  i = 2*i
        rightSon i = 2*i+1
        (vecPermutation, rowIndx, quadPermutation, colIndx)
          |north && west  = (vecIdentity, leftSon,  ident,        leftSon )
          |north          = (vecIdentity, leftSon,  colExchange,  rightSon)
          |west           = (vecExchange, rightSon, rowExchange,  leftSon )
          |otherwise      = (vecExchange, rightSon, diagExchange, rightSon)
        (a_nw, a_ne, a_sw, a_se)                = quadPermutation aQuads
        (s_nw, s_ne, s_sw, s_se) |s == ZeroM  = (ZeroM,ZeroM,ZeroM,ZeroM)
                                 |otherwise   = quadPermutation sQuads where
                                                (Mtx sQuads) = s
        (a_nw, a_ne, a_sw, a_se)             = quadPermutation aQuads
        (a'_nw, minus_v, s'_nw, gauss_n, i,j, det) = piv depth a_nw s_nw
        (a'_sw,          s'_sw, gauss_s)     = col a_sw s_sw a_nw i j minus_v
        (a'_ne,          s'_ne)              = row a_ne s_ne gauss_n i
        a'_se                                = off a_se gauss_s a_ne i
    in (decorate  (quadPermutation (a'_nw, a'_ne, a'_sw, a'_se)),
        minus_v,
        normalize (quadPermutation (s'_nw, s'_ne, s'_sw,  s_se )),
        case (gauss_n, gauss_s) of (ZeroV,ZeroV) -> ZeroV
                                   (north,south) -> vecPermutation north south
        rowIndx i, colIndx j, det)
```

Fig. 13. HASKELL code for elimination from the PIV quadrant.

that points to one of its four subtrees and gives a depth, identifying the direction and distance to its preferred pivot block. The decorations will have been installed as such a matrix is built, bottom-up. As a result, any decorated matrix "knows" which block (stripe and colonnade) is to be eliminated next; the decoration at its root even knows how big it is.

So `piv` only takes three arguments, a depth, a decorated matrix A and a disjoint, undecorated matrix S. Its results are

- A', the Shur complement of A with respect to the eliminated block;
- $-V$, where V is the inverse of the signposted pivot block, as the right factor to the Gauss vector;

- S' which is the disjoint sum of S and the eliminated stripe and colonnade;
- a Gauss vector, the pivot colonnade, abstracted from S'.
- i and j, reverse-Ahnentafel indices to the stripe and colonnade;
- the determinant of the signposted pivot block.

Of these, the second and fourth are unnecessary results from a pivot step at its outermost call; they are necessary only to a `piv` recurrence. In context, Algorithm 2.1 applies `piv` repeatedly to A_l and S_l until the latter is $0 = A_k$. At that point the interesting results are S_k; the sequences: $\langle i_l \rangle_{l=1\ldots k}$ and $\langle j_l \rangle_{l=1\ldots k}$, identified as Π and Ψ; and the cumulative product of all the pivots' determinants, which is the determinant of the original matrix, but for its sign.

The code for `piv` follows Algorithm 2.1. The base case – when the depth is zero – is simply a call to an `invert` function that inverts a scalar, s, directly to $\langle 1/s, s \rangle$ or invokes Algorithm 2.6 on non-trivial matrices. (This may cause an indirect recursion to `piv`.) When the depth is properly positive, `piv` inspects the signpost at the root and extracts *only* direction. From that it selects one of four quadrant permutations and one of two binary permutations that collapse the four possible orientations illustrated at the bottom of Fig. 7 to the leftmost one there, where the PIV block is northwest. These permutations are all self-inverting so, when the four quadrant computations are done, they will be applied again to restore the original orientation.

Then, `piv` is applied recursively to the northwest PIV quadrant, `row` to the northeast ROW quadrant; `col` to the southwest COL quadrant; `off` to the southeast OFF quadrant. The data dependencies under strict evaluation suggest that the northwest work is completed, then northeast and southwest can be done in parallel, and southeast follows the completion of northeast. Through lazy evaluation [12], however, one can imagine northwest and northeast eliminations proceeding almost simultaneously, with southeast and southwest lagging not too far behind; depending on communication patterns, all four can advance at once.

3.2. Padding via permutation

However, the P and Q described in Section 2 are not quite what we want for subsequent use of the quadtree decomposition. Before permuting with P and Q, notice that we would like the quasidiagonal blocks to land as indexed subtrees in the quadtree representation of $L + U'$ (alternatively, as proper substructures of another decomposable matrix structure.) This is not so, for instance, in Fig. 3 or in Fig. 4, where the 2×2 quasidiagonal block in S_3 has no index within S_3. The remedy is fairly easy.

As the P, Q permutations are computed in Algorithm 2.2, they will be expanded to P', Q', also meeting this criterion: each I block in P' and Q', particularly those associated with a pivot step, must land in a stripe and in a colonnade that has an Ahnentafel index. This adjustment is done to assure that applying the new permutations P' and Q' preserves the property that eliminated blocks occur intact as subtrees: both in A and, as quasidiagonal blocks, in $L + U'$.

Theorem 3.1. *Consider the I entries that are subtrees in the quadtree representation of a permutation P. Then the blocks of S selected by each of those entries will also have Ahnentafel indices into both PS and SP.*

Proof. By induction on the structural decomposition of the quadtree structure for P. When either factor is a permutation matrix, the code in Fig. 2 for matrix product never reaches a scalar basis and never adds two non-zero terms. □

Starting from Π and Ψ from Algorithm 2.1, Π' and Ψ' are computed in Algorithm 2.2 to meet a criterion of "balance" that is defined below. Then P' and Q' will have internal padding so that

$$L + U' = P' \begin{pmatrix} S & 0 \\ 0 & I \end{pmatrix} Q'$$

meets the criterion above.

The adjusted $L + U'$ matrix may be larger, and will have more interior $0, I$ entries in it, but they will follow a pattern like permutation matrices': such an I entry forces 0 entries in every other position in both its stripe and its colonnade.

While more space may be necessary, much of the padding will collapse to 0 subtrees high in the tree (with low index). The collapse assures little computational burden for the sequel Algorithms 2.4, 2.5, and 2.6, because they already respond to $0, I$ padding with immediate results: identities and annihilators. And because the permutations are used only twice (once in Algorithm 2.2 and either at Steps 2,4 of Algorithm 2.3, at Step 3 of Algorithm 2.5, or at Step 2 of Algorithm 2.6) increasing the order of the problem causes little rearrangement beyond that already necessary to bring it to $L + U'$ form.

Definition. A vector, represented as a binary tree, is balanced when the sum of all its elements is a power of two, and either it is order-1 or it has two subvectors that are also balanced.

Balancing Ω (and rearranging Π, Ψ consistently) can be accomplished by doubling its order, and padding null information within it. If the order of the matrix must be doubled $p > 0$ times to provide padding to balance Ω, then Theorem 1.5 applies each time to correct the indices in Π and Ψ. Thus, each entry, i, in either Π or Ψ maps to $2^m i$. Padding indices will be odd, either of the form $2^p i + 1$ or less than $n2^p$, where n is the order of the underlying problem.

For example, Fig. 4 has an unbalanced $\Omega_3 = \langle 1, 2, 1 \rangle$, but we can pad it to a balanced

$$\Omega' = \langle 1, \mathbf{1}, 2, 1, \mathbf{1}, \mathbf{2} \rangle \cong [[[1\ \mathbf{1}]\ 2][[1\ \mathbf{1}]\ \mathbf{2}]].$$

$$P' = \begin{pmatrix} 0 & I\ 0 & 0\ 0 & 0 \\ & 0\ 0 & \mathbf{I}\ 0 & \\ \mathbf{I} & 0 & 0 & 0 \\ 0 & 0\ I & 0\ 0 & 0 \\ & 0\ 0 & 0\ \mathbf{I} & \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} \end{pmatrix}; \quad \begin{pmatrix} I\ 0 & 0 & 0\ 0 & \mathbf{0} \\ 0\ 0 & & I\ 0 & \\ 0 & \mathbf{I} & 0 & \mathbf{0} \\ 0\ \mathbf{I} & 0 & 0\ 0 & \mathbf{0} \\ 0\ 0 & & 0\ \mathbf{I} & \\ 0 & 0 & 0 & \mathbf{I} \end{pmatrix} = Q'.$$

$$L+U' = P' \begin{pmatrix} S_3 & 0 \\ 0 & I \end{pmatrix} Q' = \left(\begin{array}{cccc|c} 1/4\ 0 & 4 & 1 & 3 & 0\ \mathbf{0} \\ 0\ \mathbf{I} & 0 & 0 & 0 & 0 \\ \hline -3/4\ 0 & -1 & -1/5 & -5/4 & 0\ \mathbf{0} \\ -1/4\ 0 & 0 & -4/5 & 1/4 & 0 \\ \hline -1\ 0 & -1 & -1/5 & 5 & 0\ \mathbf{0} \\ 0\ 0 & 0 & 0 & 0\ \mathbf{I} & \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} & \end{array}\right).$$

$$I-L = \left(\begin{array}{cccc|c} I & & & & \\ 0\ \mathbf{I} & & & & \\ \hline 3/4\ 0 & & & & \\ 1/4\ 0 & & \mathbf{I} & & \\ \hline 1\ 0 & 1 & 1/5 & I & \\ 0\ 0 & 0 & 0 & 0\ \mathbf{I} & \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} & \end{array}\right); \quad \left(\begin{array}{cccc|c} 4\ 0 & 4 & 1 & 3 & 0\ \mathbf{0} \\ \mathbf{I} & 0 & 0 & 0 & 0 \\ \hline & -1 & 1/4 & -5/4 & 0\ \mathbf{0} \\ & 0 & -5/4 & 1/4 & 0 \\ \hline & & & 1/5 & 0\ \mathbf{0} \\ & & & \mathbf{I} & \\ \hline & & & & \mathbf{I} \end{array}\right) = U.$$

Fig. 14. Another permutation of rational triangulation in Fig. 4.

The insertions appear in boldface. Compatible versions of Π and Ψ are

$$\Pi' = \langle 10, \mathbf{9}, 4, 14, \mathbf{13}, \mathbf{7} \rangle \cong [[[10\ \mathbf{9}]\ 4][[14\ \mathbf{13}]\ \mathbf{7}]],$$

$$\Psi' = \langle 8, \mathbf{9}, 6, 12, \mathbf{13}, \mathbf{7} \rangle \cong [[[8\ \mathbf{9}]\ 6][[12\ \mathbf{13}]\ \mathbf{7}]].$$

The resulting P' and Q' appear in Fig. 14, where padding is also in boldface.

Efficient algorithms have been written to balance, perhaps expanding, Ω and to construct the corresponding Π' and Ψ', but they are not of great interest here. The choice of the extended P' and Q' can be determined from Π and Ψ, alone, in time proportional to n, the order of the underlying problem.

Since we ordinarily expect n to be a power of two, the order of its quadtree representation, A, will already have been padded by $2^{\lceil \lg n \rceil} - n$, anywhere from 0% up to 100% of n. This zero padding (initially all to the southeast) can be used to balance Ω by permuting A's representation in Algorithm 2.2 so that that same padding becomes internal. Coarse analysis on random Ω suggests that an average balanced order would be $1.5n$; so, in 33% of cases Ω could be balanced without increasing $2^{\lceil \lg n \rceil}$; i.e., $p = 0$ would often suffice in the description above. It also suggested that other cases of balancing would rarely double n ($p = 1$, adding one level to the quadtree). Actual experience, which is not random but skewed by pivot-selection strategies, indicates that even such doubling, in fact, happens very rarely, indeed.

3.3. Pivot selection

Pivoting is the search for the next block to be eliminated; such a search becomes necessary whenever zero blocks can interfere with local inversion and may even be desirable, in any event, to improve behavior of the triangulation algorithm. Usually the search occurs properly between elimination steps but, because we are already manipulating trees – the ideal structure for searching, it is easy to distribute the search backwards through the previous elimination step. The key idea, borrowed from generic search algorithms, is to stash information at the interior nodes of the quadtree.

Although pivoting usually deals with blocks of only one size, we are dealing with elimination candidates of various sizes. Size and other comparative information is installed at interior nodes bottom-up, as a decorated tree is built. The function `decorate` (cf. Fig. 8) not only normalizes the four subtrees, but also extrapolates local information from *their* decorations. Decorations costs space, but they save memory reads because the arguments to `decor` in Fig. 8 are likely to be already available in cache. Pivoting later would require additional fetches from memory to retrieve that information.

The description here reads as if complete pivoting [17, Section 3.4.8] is used. Partial pivoting is also possible, but it has no particular advantage for the symmetric, tree structure. The uneliminated portions of the matrix A_l of Algorithm 2.1 are represented as a decorated matrix in the code of Section 3.1. Details of the decoration depend on the pivot strategy, but all decorations include a signpost (Fig. 12).

Definition. A *signpost* is a triple composed of two boolean values and a tiny integer.

Definition. A *decoration* is a composed of a signpost and other atomic information.

Definition. A *decorated* matrix is the quadtree representation of a matrix with a decoration on each subtree.

The signpost points toward the block within each subtree that is the locally preferred choice to be eliminated next. The two boolean values select north/south, east/west; its integer indicates the depth to that block, and can be "tiny" because it is bounded by the base-2 logarithm of the order of the problem. Every non-zero matrix contains at least one candidate for elimination, even if it is only a scalar at the remotest leaf.

3.3.1. Decorating to preserve stability

If decorations were to identify the largest (in magnitude) scalar element that had not yet been eliminated (to assure the smallest possible multipliers in the Gauss transform [17], of Figs. 9 and 10), then it would suffice that each decoration also include the maximum magnitude of all scalars in that quadrant. That is, `More` in Fig. 12 would be the type of a scalar, `a`. Then `decor` becomes little more than `max` applied to the four local maxima, or the four absolute values when the quadrants are scalars.

Our goal of identifying pivot blocks of different sizes, however, requires that some non-singular blocks also be identified as candidates for elimination. To that end, the following semidecision algorithm is offered.

Definition. A matrix of order 2^p is nown-singular if $p < 2$ and it is non-singular, or if $p = q + 1$, one of its four quadrants (of order 2^q) is 0, and the two adjacent quadrants are nown-singular.

The term, "nown-singular", is chosen to suggest that, under this matrix representation, such blocks are easily *known* to be *non*-singular. However, not every uneliminated, non-singular quadrant will be nown-singular and, therefore, a pivot candidate. This is a fast, but partial, filter for candidate elimination-blocks. Any non-zero matrix can be decorated to identify some nown-singular subblock as the locally preferred pivot.

This definition provides base cases of 1×1 and 2×2. Alternatively, a 3×3 array might be defined as the scalar type; all its primitive operations are also easily computed.

Theorem 3.2. *Every nown-singular matrix is non-singular.*

Theorem 3.3. *Every non-singular triangular matrix is nown-singular.*

Corollary 3.1. *Every non-singular,* (2×2)*-block-triangular matrix is nown-singular.*

Theorem 3.4. *The inverse of a nown-singular matrix is nown-singular.*

Definition. The *known-determinant* of a matrix is the magnitude of its determinant if it is nown-singular, and zero otherwise.

Although computing a known-determinant is easy, its magnitude is not directly used for stability of undulant pivoting. The problem is that, for instance, the known-determinant of a 4×4 block cannot be compared with that of a 1×1 in this context. Instead, the geometric mean of the magnitudes of the scalar pivots in the block is proposed as a weighted measure.

3.3.2. Decorating for parallelism and fill-in

Decorations have been introduced to control stability, but pivoting can have other goals. Eliminating a larger block in a single step improves both locality of computation and macroscopic, parallel behavior. With sparse matrices, good pivoting will control fill-in. And, of course, all these goals may be blended.

Eliminating larger blocks improves performance in two ways. First, it improves locality, because the base conditions (when $i = 1$) of `row`, `col`, `off`, as well as f, g, h, encapsulate computation that would require more communication under scalar elimination; better locality reduces page and cache misses on uniprocessors, and similarly reduces probes of remote memory on multiprocessors. Second, for instance, when a

2×2 block is eliminated in one step, the overhead for a single process dispatch in a parallel Shur complement can be amortized against twice the progress as for a 1×1 step. So the net overhead of parallel processing is reduced by eliminating larger blocks, in proportion to their sizes.

In order to choose a larger pivot block, `decor` need only favor shallower blocks; signposts already contain the necessary information.

Algorithms for sparse matrices often use a "Markowitz count" [9, Section 9.2] to avoid fill-in. It predicts the fill caused by eliminating any candidate, based on two vectors: the "row counts" and "column counts", whose size is the order of the matrix. These can be extended to the analogous *stripe counts* and *colonnade counts* with reverse-Ahnentafel indices, and used to compute quickly an analogous prediction for decorating every block pivot. Unlike other decorations, however, these numbers can change as a result of elimination elsewhere in the matrix; that is, such decorations go stale without traversing the decorated submatrix. Experiments with such decorations [3, Section 4.4] have, nevertheless, yielded very good results.

3.3.3. Decorating to avoid bignums

The problem of the next section suggests an opposite use of the known-determinant, as introduced above. Exact-arithmetic GE can generate huge integers in the intermediate results even though the final inverse may be quite tame. Elements of the underlying domain may be symbolic formulae, or simply unbounded integers: bignums (in LISP's jargon) or Integers (in HASKELL's). Ordinary operations like addition, multiplication, and especially division are slowed severely on the internal representations of larger elements.

As described in the next section, integer GE can avoid these large numbers by choosing to eliminate blocks whose determinant is unitary or, more practically, whose non-zero known-determinant is as *small* in magnitude as possible. That is, the 5-way Wilkinson-`max` for decorating floating-point matrices (4 subtrees plus 1 enclosing tree) becomes a 5-way `min` for decorating integer matrices – excluding, of course, zero trees.

4. Exact-arithmetic decomposition

> Simplicity does not precede complexity, but follows it. [18, ¶31]

4.1. Integer-matrix inversion

Integer solutions for linear systems are important in rational arithmetic and symbolic computation. Succinctly stated, the problem is to compute from an integer matrix A both $d = \det A$ and dA^{-1}. That the latter is, itself, an integer matrix is immediately recalled from (the much less efficient) Cramer's rule. In order to produce exact answers only integer operators may be used; all intermediate results are precise, often

requiring extended-integer representations that raise the relative costs for multiplication and, especially, for division.

The usual algorithm to solve this problem is due to Bareiss. A perspective at the end of his introduction [2, p. 567] characterizes the quadtree algorithm below. It is a fraction-free version of GE that eliminates blocks of undulant size: $1, 2, 4, \ldots$ with quadtree matrices that Bareiss would recognize each as a 2^p-step. Although he describes a version with invariant p, he explicitly suggests degrees higher than the two-step algorithms that he presents [2, p. 570]. That is, Bareiss anticipates the parallel nature of his elimination steps that is motivational here.

The quadtree solution, however, brings new attributes to this problem. Implicitly, it offers a uniform method for both sparse and dense problems that implicitly responds to sparseness. The parallel decomposition is married to a structural decomposition, as well. The tree structure offers good pivoting strategies, a subject treated only briefly by Bareiss [2, Section V]. Keeping the accumulated determinants small can be as important as sustaining sparseness; both constrain the extra time and space needed for the residual problem. Finally, the following algorithm provides undulant pivoting, allowing better pivot selection and even more parallelism, if only at a few intermediate steps.

The algorithm below, an integer-preserving version of Algorithms 2.2 and 2.6, is closest to Bareiss's fraction-free, multistep techniques. The elementary operations are integer addition, multiplication, and exact division (with selectively small divisors). It is desirable to avoid division and to keep the magnitude of intermediate results as small as possible because larger arguments retard these elementary operations. For that reason, the pivoting strategies of Sections 3.3.2 and 3.3.3 are most welcome.

4.2. Integer-matrix decomposition

Notation. *Integer matrices are denoted with "barred" identifiers, like* $\bar{A}$.

As before, the integer k is used globally to indicate the number of elimination steps, but locally as the order of $\bar{A}_{mc}$.

Under undulant elimination, the latter meaning for $k = o_l$ takes values that vary from Step l to the next. Recalling that in $L + U'$ decomposition, $\Omega = \langle o_1, \ldots, o_k \rangle$ is the list of these k orders, which prescribe the quasi-diagonal portion of $L + U'$, we introduce two more lists of the same length for use with integer-matrix decomposition.

Notation. $\mathrm{T} = \langle t_1, t_2, \ldots, t_k \rangle$ *is a list of the same length at* Ω, *where* t_l *is the determinant of the* l^{th} $(o_l \times o_l)$ *pivot block, which appears as* $\bar{A}_{l,mc}$ *below.*

The letter 'Tau' is taken from the last letter in "determinant". An alternative formulation of T is the diagonal matrix $\bar{T}$: as a quasidiagonal matrix, each block of order o_l along its main diagonal is a scalar matrix of the integer t_l.

Notation. *$\Delta = \langle d_0, d_1, \ldots, d_{k-1} \rangle$ is a list of the same length at Ω, where d_l is the determinant of the portion of $\bar{A}$ that has been eliminated by the first l elimination steps in the decomposition of $\bar{A}$; $d_0 = 1; d_k = \det \bar{A}$.*

The letter "Δ" is derived from the first letter in "*d*eterminant" but, so not to confuse with diagonal or quasidiagonal ones, the analogous matrix is awkwardly called $\bar{R}$. The diagonal matrix $\bar{R}$ is another formulation of Δ: viewed as a quasidiagonal matrix, each block of order o_l along similarly to $\bar{R}$'s main diagonal is a scalar matrix of the integer d_{l-1}.

Definition. An $\bar{L} + \bar{U}'$ *decomposition* of $\bar{A}$, non-singular integer matrix of order n, is the septuple, $\langle d, \bar{L} + \bar{U}', \Delta, \Omega, \mathrm{T}, P, Q \rangle$ where

- $\langle d_k, L + U', \Omega, P, Q \rangle$ is an $L + U'$ decomposition of $\bar{A}$;
- $\Delta = \langle d_0, d_1, \ldots, d_{k-1} \rangle$ and $\mathrm{T} = \langle t_1, t_2, \ldots, t_k \rangle$ are lists of integers of the same length as $\Omega = \langle o_1, o_2, \ldots, o_k \rangle$;
- $d_0 = 1$; $\forall l (d_{l+1} = t_{l+1} d_l^{1-o_l})$;
- Δ prescribes the diagonal matrix $\bar{R}$, whose k quasidiagonal blocks are scalar matrices with orders given by Ω;
- T prescribes the diagonal matrix $\bar{T}$ whose k quasidiagonal blocks are scalar matrices with orders from Ω.
- $\bar{L} = L\bar{T}$, an integer matrix with L as specified in the definition of a $L + U'$ decomposition of $\bar{A}$.
- $\bar{U} = \bar{R}U$, an integer matrix with U specified similarly.
- $\bar{D} = \bar{T}D\bar{R}^{-1}$, an integer matrix with D specified similarly.
- $\bar{U}' = \bar{U} + \bar{D} - \bar{R}D^{-1}$, an integer matrix.

Corollary 4.1. *In an $\bar{L} + \bar{U}'$ decomposition, $P\bar{A}Q = (I - \bar{L}\bar{T}^{-1})\bar{R}^{-1}\bar{U}$.*

Just as this definition depends on the definition of $\bar{L} + \bar{U}'$ decomposition, Algorithms 4.1 and 4.2 are best explained from Algorithms 2.1 and 2.2. The following invariant is stated here to bridge this definition to the new algorithm.

Invariant. *After the lth elimination step of Algorithm* 4.1 *these scalar-matrix products relate A_l, S_l of Algorithm* 2.1 *to $\bar{A}_l, \bar{S}_l$ of Algorithm* 4.1, *using identical pivoting.*

1. *d_l is the determinant of the eliminated (northwest) portion of $P_l\bar{A}Q_l$ for partial permutations P_l, Q_l abstracted from Π_l, Ψ_l.*
2. $\bar{S}_{l,mc} = t_l S_{l,mc} / d_{l-1}$;
3. $\bar{S}_{l,mw} = d_{l-1} S_{l,mw}$; $\bar{S}_{l,me} = d_{l-1} S_{l,me}$;
4. $\bar{S}_{l,nc} = S_{l,nc} t_l$; $\bar{S}_{l,sc} = S_{l,sc} t_l$;
5. $\bar{A}_l = d_l A_l$.

Algorithm 4.1. *Integer triangular decomposition* of non-singular $\bar{A}$ of order n to $\langle d, \bar{S}, \Delta, \Omega, \mathrm{T}, \Pi, \Psi \rangle$, for $\bar{L} + \bar{U}'$ decomposition.

The decomposition parallels Algorithm 2.1. Complete, undulant-block pivoting is again assumed. Starting from the initial tuple $\langle \bar{A}, 1, 0, [\,], [\,], [\,], [\,], [\,] \rangle$, the following elimination step from one octuple $\langle \bar{A}_l, d_l, \bar{S}_l, \Delta_l, \Omega_l, \mathrm{T}_l, \Pi_l, \Psi_l \rangle$ to the next, $\langle \bar{A}_{l+1}, d_{l+1}, \bar{S}_{l+1}, \Delta_{l+1}, \Omega_{l+1}, \mathrm{T}_{l+1}, \Pi_{l+1}, \Psi_{l+1} \rangle$ finally results in $\langle 0, d, \bar{S}, \Delta, \Omega, \mathrm{T}, \Pi, \Psi \rangle$.

Let the block decomposition of $\bar{A}$, isolating the $k \times k$ invertible pivot block, $\bar{A}_{mc}$, be labeled

$$\bar{A} = \begin{array}{c} \\ i \\ k \\ n-k-i \end{array} \begin{array}{c} \begin{array}{ccc} j & k & n-k-j \end{array} \\ \begin{pmatrix} \bar{A}_{nw} & \bar{A}_{nc} & \bar{A}_{ne} \\ \bar{A}_{mw} & \bar{A}_{mc} & \bar{A}_{me} \\ \bar{A}_{sw} & \bar{A}_{sc} & \bar{A}_{se} \end{pmatrix} \end{array},$$

where $\bar{A}$ is $n \times n$ and $\bar{A}_{nw}$ is $i \times j$. (Again, the local indices i, j, k are determined by pivoting.) The order of $\bar{A}_{mc}$ is $k = o_{l+1}$.

Then decompose $\bar{S}_l$ similarly, reflecting elimination already done:

$$\bar{S}_l = \begin{array}{c} \\ i \\ k \\ n-k-i \end{array} \begin{array}{c} \begin{array}{ccc} j & k & n-k-j \end{array} \\ \begin{pmatrix} \bar{S}_{nw} & \bar{S}_{nc} & \bar{S}_{ne} \\ \bar{S}_{mw} & 0 & \bar{S}_{me} \\ \bar{S}_{sw} & \bar{S}_{sc} & \bar{S}_{se} \end{pmatrix} \end{array}.$$

As before, corresponding blocks of these two matrices are disjoint, pairwise:

$$\Omega_{l+1} = \Omega_l + +[k]; \quad \Pi_{l+1} = \Pi_l + +[\mathrm{RA}(i,k,n)]; \quad \Psi_{l+1} = \Psi_l + +[\mathrm{RA}(j,k,n)].$$

Then invert $\bar{A}_{mc}$ to yield both $t = \det \bar{A}_{mc}$ and $\bar{S}_{mc} = t\bar{A}_{mc}^{-1}$ as a result of a single, integer-preserving (perhaps recursive) computation. In the trivial case that $k = 1$, $\bar{S}_{mc} = 1$ and $t = \bar{A}_{mc}$:

$$\begin{aligned} d_{l+1} &= t/d_l^{k-1}; \quad \Delta_{l+1} = \Delta_l + +[d_l]; \quad \mathrm{T}_{l+1} = \mathrm{T}_l + +[t], \\ \begin{pmatrix} \bar{G}_n \\ \bar{G}_s \end{pmatrix} &= \begin{pmatrix} \bar{A}_{nc} \\ \bar{A}_{sc} \end{pmatrix} (-\bar{S}_{mc}), \\ \bar{S}_{l+1} &= \begin{pmatrix} \bar{S}_{nw} & \bar{S}_{nc} \oplus \bar{G}_n & \bar{S}_{ne} \\ \bar{S}_{mw} \oplus \bar{A}_{mw} & \bar{S}_{mc} & \bar{S}_{me} \oplus \bar{A}_{me} \\ \bar{S}_{sw} & \bar{S}_{sc} \oplus \bar{G}_s & \bar{S}_{se} \end{pmatrix}, \\ \bar{A}_{l+1} &= \left[t \begin{pmatrix} \bar{A}_{nw} & 0 & \bar{A}_{ne} \\ 0 & 0 & 0 \\ \bar{A}_{sw} & 0 & \bar{A}_{se} \end{pmatrix} + \begin{pmatrix} \bar{G}_n \\ 0 \\ \bar{G}_s \end{pmatrix} \begin{pmatrix} \bar{A}_{mw} & 0 & \bar{A}_{me} \end{pmatrix} \right] \Big/ d_l^k. \qquad \square \end{aligned}$$

Fig. 15 illustrates this algorithm with example in Fig. 4 using the identical pivoting. The $(l+1)$st elimination step is division-free if $d_l = 1$. In the following k, from above, becomes o_{l+1}, and t is similarly t_{l+1}; so $\Delta, \Omega, \mathrm{T}$ are indexed consistently.

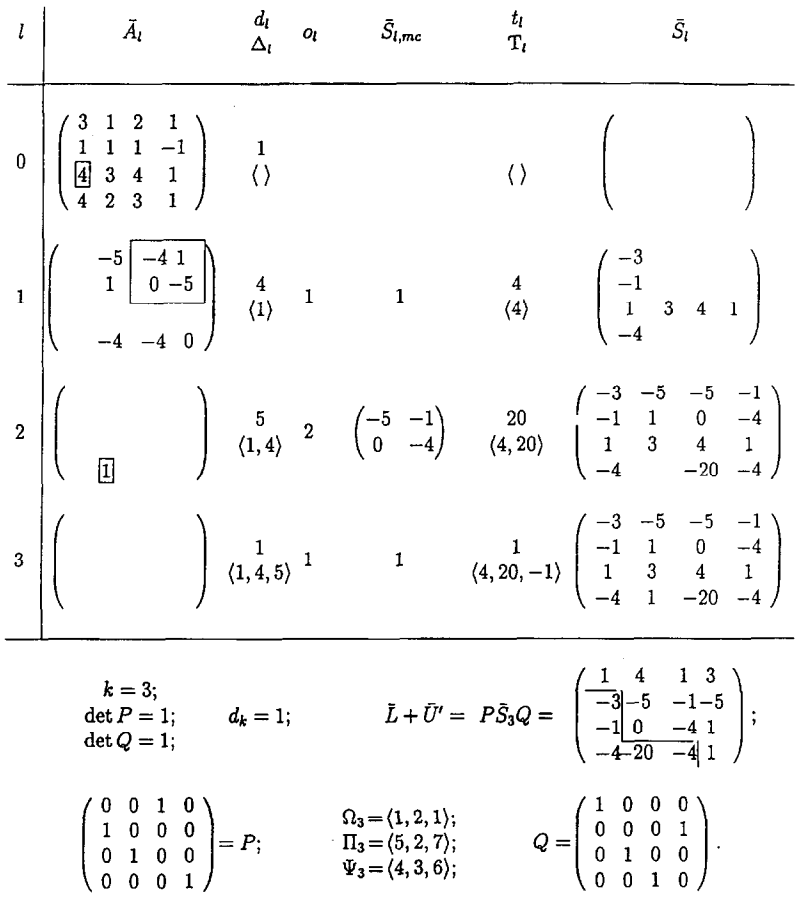

Fig. 15. Integer triangulation using same pivoting as in Fig. 4.

Algorithm 4.2 ($\bar{L} + \bar{U}'$ *decomposition*). Compute the integer triangular decomposition of $\bar{A}$: $\langle d, \bar{S}, \Delta, \Omega, \mathrm{T}, \Pi, \Psi \rangle$ using Algorithm 4.1. Then use the permutations P, Q associated with Π, Ψ to obtain $\langle d(\det P)(\det Q), P\bar{S}Q, \Delta, \Omega, \mathrm{T}, P, Q \rangle$. as in Algorithm 2.2. □

Theorem 4.1. *Algorithm* 4.1 *maintains the Invariant stated just before it.*

Proof. It is important to establish that all divisions are exact; this follows in all cases from the observation that the partial result from rational triangulation (Algorithm 2.2) is multiplied by the appropriate scalar (d_l or t_{l+1}) to assure that the result is an exact integer.

Satisfying the invariant for l will be shown sufficient to establish it for $l + 1$. The argument reads most simply if we presume, as in the proof of Theorem 2.2, that no pivoting were necessary – that the permutations P, Q have been given a priori and already applied to A; so all the north and west submatrices are void.

Assume that d_l is the determinant of the portion of $P\bar{A}Q$ that has been eliminated through the lth elimination step; it is necessarily an integer because $\bar{A}$ is an integer matrix.

1. Since $\bar{A}_l = d_l A_l$,

$$t_{l+1} = \det \bar{A}_{mc} = \det(d_l A_l) = d_l^{o_{l+1}} (\det A_{mc});$$

$$d_{l+1} = d_l (\det A_{mc}) = t_{l+1} d_l^{1-o_{l+1}}.$$

2. $\bar{S}_{l+1,mc} = t_{l+1} \bar{A}_{l,mc}^{-1} = t_{l+1} (d_l A_{l,mc})^{-1} = t_{l+1} A_{l,mc}^{-1} / d_l = (t_{l+1} S_{l,mc})/d_l$.
3. Considering only those (off-diagonal) entries in the pivot stripe that contribute a stripe to $\bar{U}$:

$$\bar{U}_{l+1,me} = \bar{A}_{l,me} = d_l A_{l,me} = d_l S_{l+1,me}.$$

4. Considering, similarly, only those contributing to $\bar{L}$:

$$\bar{L}_{l+1,sc} = \bar{A}_{l,sc}(-\bar{S}_{l+1,mc}) = (d_l A_{l,sc})(-t_{l+1} S_{l,mc}/d_l) = S_{l+1,sc} t_{l+1}.$$

5. Considering only the uneliminated (southeast) portion of $P\bar{A}Q$:

$$\begin{aligned}
\bar{A}_{l+1,se} &= (t_{l+1} \bar{A}_{l,se} + \bar{S}_{l+1,sc} \bar{A}_{l,me}) / d_l^{o_{l+1}} \\
&= [t_{l+1}(d_l A_{l,se}) + (t_{l+1} S_{l+1,sc})(d_l A_{l,me})] / d_l^{o_{l+1}} \\
&= [t_{l+1} d_l^{1-o_{l+1}}][A_{l,se} + S_{l+1,sc} A_{l,me}] \\
&= d_{l+1} A_{l+1,se}. \qquad \square
\end{aligned}$$

Corollary 4.2. *Algorithm* 4.2 *computes the* $\bar{L} + \bar{U}'$ *decomposition of integer matrix* $\bar{A}$.

Proof. Follows that of Theorem 2.2 with a simple induction on l of Algorithm 4.1. The invariant is satisfied for $l = 0$ since $\bar{A} = \bar{A}_0$ and $d_0 = 1$. Since $\bar{A}$ is non-singular, Algorithm 2.1 would successfully decompose it in k steps (resulting in a rational S_k.) The induction uses Theorem 4.1 to establish the invariant on $\bar{S}_k$ and the ultimate diagonal matrices, $\bar{R}$ and $\bar{T}$ prescribed by Δ_k and T_k, respectively stripe-by-stripe and colonnade-by-colonnade. That is, Invariant 2 assures that $\bar{T} D \bar{R}^{-1} = \bar{D}$; Invariant 4 establishes $\bar{L} = L\bar{T}$. Invariant 3 implies $\bar{U}' - \bar{D} = \bar{R}(U - D^{-1})$; so $\bar{U}' = \bar{U} + \bar{D} - \bar{R} D^{-1}$. $\square$

Again, although these algorithms have been described abstractly for any block representation, ours uses the quadtree matrix representation. The codes appear very similar to those presented in Section 3.1, except for the extra parameters of d and t, and their resulting lists: Δ and T.

4.3. Permuting $\bar{S}$ to $\bar{L} + \bar{U}'$

After decomposition, the discussion of Section 3.2 applies. Because the elimination in Fig. 15 follows the same geography as in Fig. 4, the example Ω', Π', Ψ' in Section 3.2 apply, as well, to the sample decomposition in Fig. 15.

As Ω, Π, Ψ are stretched, however, we must stretch and pad Δ and T (and, by implication, $\bar{R}$ and $\bar{T}$.) The values to pad there depend on the sequel algorithm that uses them to compute $d\bar{A}^{-1}$, below.

It turns out that T can be padded with zeroes as it is stretched, because its padding is never used. Δ however, contains meaningful determinants; the entry *following* a stretch should be replicated to pad it. The entry that follows, or $d = \det A$ when the padding is suffix, is required padding because the indexing of Δ is shifted relative to that for T.

Therefore, our example would be padded to an 8×8 matrix with the following lists and vectors:

$$\Delta' = \langle 1, \mathbf{4}, 4, 5, \mathbf{1}, \mathbf{1} \rangle \cong [[[1\ \mathbf{4}]\ 4][[5\ \mathbf{1}]\ \mathbf{1}]],$$

$$\mathrm{T}' = \langle -4, \mathbf{0}, -20, -1, \mathbf{0}, \mathbf{0} \rangle \cong [[[-4\ \mathbf{0}]\ -20][[-1\ \mathbf{0}]\ \mathbf{0}]].$$

Finally, as a demonstration that good pivoting can improve performance, Fig. 16 shows better pivoting for our example problem. It uses strategies from Sections 3.3.3 and 3.3.2 to favor elimination of blocks with unit nown-determinant, and big ones where possible. In this case, the southeast 2×2 quadrant of $\bar{A}$ is the only one of four that has a unit determinant; therefore, it is the largest subtree with the smallest nown-determinant, and the best candidate for the first pivot block. In this case, no Section 3.2 padding is necessary; the index lists can be used as shown. All denominators are units, and so all divisions become trivial. It is notable how choosing a pivot with small nown-determinant accelerates not only decomposition, but also (later) the inverting functions: $\bar{f}$, $\bar{g}$, and $\bar{h}$.

4.4. Integer-preserving inversion algorithm

Now comes the analog of Algorithm 2.6 to complete the inversion.

Algorithm 4.3 (*Integer-preserving matrix inversion*). Invert $\bar{A}$ to $\langle d\bar{A}^{-1}, d \rangle$ where $d = \det \bar{A}$.

1. Compute the $\bar{L} + \bar{U}'$ decomposition of A using Algorithm 4.2.
2. Compute $d\bar{A}^{-1} = Q[\bar{f}(d, \bar{L} + \bar{U}', \Delta, \Omega, \mathrm{T})]P$.

The three functions, $\bar{f}, \bar{g}, \bar{h}$ are defined mutually and computed recursively:

- Define $\bar{f} : (d_s, \bar{L} + \bar{U}', \Delta, \Omega, \mathrm{T}) \mapsto \bar{M}$, where $\bar{M}$ is the same size as $\bar{L} + \bar{U}'$, as follows:
  - If $\Omega = \langle o \rangle$ is of length one, then $\Delta = \langle d \rangle$ and $\bar{M} = (\bar{L} + \bar{U}')d^{2-o}$.
  - Otherwise, consistently partition

$$\bar{L} + \bar{U}' = \begin{pmatrix} \bar{L}_{nw} + \bar{U}'_{nw} & \bar{E} \\ \bar{W} & \bar{L}_{se} + \bar{U}'_{se} \end{pmatrix};$$

$$\Delta = \langle \Delta_n, \Delta_s \rangle; \quad \Omega = \langle \Omega_n, \Omega_s \rangle, \qquad \mathrm{T} = \langle \mathrm{T}_n, \mathrm{T}_s \rangle.$$

l	$\bar{A}_l$	d_l / Δ_l	o_l	$\bar{S}_{l,mc}$	t_l / T_l	$\bar{S}_l$
0	$\begin{pmatrix} 3 & 1 & 2 & 1 \\ 1 & 1 & 1 & -1 \\ 4 & 3 & \boxed{4} & 1 \\ 4 & 2 & 3 & 1 \end{pmatrix}$	1 $\langle\,\rangle$			$\langle\,\rangle$	$\begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix}$
1	$\begin{pmatrix} -1 & 0 & & \\ 5 & \boxed{-1} & & \\ & & & \\ & & & \end{pmatrix}$	1 $\langle 1\rangle$	2	$\begin{pmatrix} 1 & -1 \\ -3 & 4 \end{pmatrix}$	1 $\langle 1\rangle$	$\begin{pmatrix} & & 1 & -2 \\ & & -4 & 5 \\ 4 & 3 & 1 & -1 \\ 4 & 2 & -3 & 4 \end{pmatrix}$
2	$\begin{pmatrix} \boxed{1} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix}$	-1 $\langle 1,1\rangle$	1	1	-1 $\langle 1,-1\rangle$	$\begin{pmatrix} & 0 & 1 & -2 \\ 5 & 1 & -4 & 5 \\ 4 & 3 & 1 & -1 \\ 4 & 2 & -3 & 4 \end{pmatrix}$
3	$\begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix}$	1 $\langle 1,1,-1\rangle$	1	1	1 $\langle 1,-1,1\rangle$	$\begin{pmatrix} 1 & 0 & 1 & -2 \\ 5 & 1 & -4 & 5 \\ 4 & 3 & 1 & -1 \\ 4 & 2 & -3 & 4 \end{pmatrix}$

$k=3;$
$\det P=-1;$ $\quad d_k=1;$ $\quad \bar{L}+\bar{U}'=P\bar{S}_3Q=\left(\begin{array}{cccc} 1 & -1 & 3 & 4 \\ -3 & 4 & 2 & 4 \\ \hline -4 & 5 & \boxed{1} & 5 \\ 1 & -2 & 0 & 1 \end{array}\right);$
$\det Q=-1;$

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}=P; \qquad \begin{array}{l} \Omega_3=\langle 2,1,1\rangle; \\ \Pi_3=\langle 3,6,4\rangle; \\ \Psi_3=\langle 3,6,4\rangle; \end{array} \qquad Q=\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

Fig. 16. Integer triangulation with pivoting that favors unit blocks.

Let the first value in Δ_s be called d_n; that is, $\Delta_s=(d_n : \Delta)$ for some suffix Δ. Compute

$$\bar{K} = \bar{f}(d_s, \bar{L}_{se}+\bar{U}'_{se}, \Delta_s, \Omega_s, \mathrm{T}_s),$$
$$\bar{F} = \bar{f}(d_n, \bar{L}_{nw}+\bar{U}'_{nw}, \Delta_n, \Omega_n, \mathrm{T}_n).$$

Compute

$$\bar{H}=\bar{g}(\bar{L}_{nw}+\bar{U}'_{nw}, \bar{E}\bar{K}, \mathrm{T}_n),$$
$$\bar{J}=\bar{h}(\bar{L}_{nw}+\bar{U}'_{nw}, \bar{K}\bar{W}, \mathrm{T}_n).$$

Finally, compute $\bar{B}=\bar{g}(\bar{L}_{nw}+\bar{U}'_{nw}, d_n\bar{E}\bar{J}, \mathrm{T}_n)$ and then $(\bar{G}=d_s\bar{F}+\bar{B})/d_n$.

$$\bar{M}=\begin{pmatrix} \bar{G} & \bar{H} \\ \bar{J} & \bar{K} \end{pmatrix}.$$

- Define $\bar{g} : (\bar{L} + \bar{U}', \bar{X}, \mathrm{T}) \mapsto \bar{M}$, with $\bar{M}$ of the same size as $\bar{X}$ and compatible with $\bar{L} + \bar{U}'$ in its number of rows, as follows:
  - If $\mathrm{T} = \langle t \rangle$ is of length one, then $\bar{M} = (\bar{L} + \bar{U}')\bar{X} / -t$.
  - Otherwise, consistently partition

$$\mathrm{T} = \langle \mathrm{T}_n, \mathrm{T}_s \rangle,$$

$$\bar{L} + \bar{U}' = \begin{pmatrix} \bar{L}_{nw} + \bar{U}'_{nw} & \bar{E} \\ \bar{W} & \bar{L}_{se} + \bar{U}'_{se} \end{pmatrix}, \qquad \bar{X} = \begin{pmatrix} \bar{X}_{nw} & \bar{X}_{ne} \\ \bar{X}_{sw} & \bar{X}_{se} \end{pmatrix}.$$

Compute $\bar{J} = \bar{g}(\bar{L}_{se} + \bar{U}'_{se}, \bar{X}_{sw}, \mathrm{T}_s)$ and $\bar{K} = \bar{g}(\bar{L}_{se} + \bar{U}'_{se}, \bar{X}_{se}, \mathrm{T}_s)$. Compute $\bar{G} = \bar{g}(\bar{L}_{nw} + \bar{U}'_{nw}, \bar{X}_{nw} + \bar{E}\bar{J}, \mathrm{T}_n)$ and $\bar{H} = \bar{g}(\bar{L}_{nw} + \bar{U}'_{nw}, \bar{X}_{ne} + \bar{E}\bar{K}, \mathrm{T}_n)$. Finally, assemble

$$\bar{M} = \begin{pmatrix} \bar{G} & \bar{H} \\ \bar{J} & \bar{K} \end{pmatrix}.$$

- Define $\bar{h} : (\bar{L} + \bar{U}', \bar{X}, \mathrm{T}) \mapsto \bar{M}$, with $\bar{M}$ of the same size as $\bar{X}$ and compatible with $\bar{L} + \bar{U}'$ in its number of columns, as follows:
  - If $\mathrm{T} = \langle t \rangle$ is of length one, then $M = \bar{X}/t$.
  - Otherwise, consistently partition

$$\mathrm{T} = \langle \mathrm{T}_n, \mathrm{T}_s \rangle;$$

$$\bar{L} + \bar{U}' = \begin{pmatrix} \bar{L}_{nw} + \bar{U}'_{nw} & \bar{E} \\ \bar{W} & \bar{L}_{se} + \bar{U}'_{se} \end{pmatrix}, \qquad \bar{X} = \begin{pmatrix} \bar{X}_{nw} & \bar{X}_{ne} \\ \bar{X}_{sw} & \bar{X}_{se} \end{pmatrix}.$$

Compute $\bar{H} = \bar{h}(\bar{L}_{se} + \bar{U}'_{se}, \bar{X}_{ne}, \mathrm{T}_s)$ and $\bar{K} = \bar{h}(\bar{L}_{se} + \bar{U}'_{se}, \bar{X}_{se}, \mathrm{T}_s)$. Compute $\bar{G} = \bar{h}(\bar{L}_{nw} + \bar{U}'_{nw}, \bar{X}_{nw} + \bar{H}\bar{W}, \mathrm{T}_n)$ and $\bar{J} = \bar{h}(\bar{L}_{nw} + \bar{U}'_{nw}, \bar{X}_{sw} + \bar{K}\bar{W}, \mathrm{T}_n)$. Finally, assemble $\bar{M}$ as in the definition of $\bar{g}$, above. □

Theorem 4.2. *Under the relationships among*

$$L, \bar{L}, U, \bar{U}, U', \bar{U}', D, \bar{D}, \bar{R}, \bar{T}, d_s, \Delta, \Omega, \mathrm{T}$$

and among X, E, W *and* $\bar{X}, \bar{E}, \bar{W}$ *implied by the definition of* $\bar{L} + \bar{U}'$ *decomposition and used in the respective definitions of* f, g, h *and* $\bar{f}, \bar{g}, \bar{h}$ *above*:

$$\bar{f}(d_s, \bar{L} + \bar{U}', \Delta, \Omega, \mathrm{T}) = d_s f(L + U', \Omega) = d_s U^{-1}(I - L)^{-1},$$

$$\bar{g}(\bar{L} + \bar{U}', \bar{X}, \mathrm{T}) = -U^{-1}\bar{R}^{-1}\bar{X},$$

$$\bar{h}(\bar{L} + \bar{U}', \bar{X}, \mathrm{T}) = \bar{X}\bar{T}^{-1}(I - L)^{-1}.$$

Proof. By mutual, course-of-values induction on the length of T. The proof is essentially a rehearsal of that for Theorem 2.3.

Bases: If $\Omega = \langle o \rangle$, then $\Delta = \langle d \rangle$; $\mathrm{T} = \langle t \rangle$; as used $d_s = td^{1-o}$; $\bar{L} = 0 = L$; and

$$\begin{aligned} \bar{f}(d_s, \bar{L} + \bar{U}', \Delta, \Omega, \mathrm{T}) &= (\bar{L} + \bar{U}')d^{2-o} = \bar{D}d^{2-o} = \bar{T}D\bar{R}^{-1}d^{2-o} \\ &= tDd^{1-o} = d_s D = d_s U^{-1}(I - 0)^{-1} \\ &= d_s U^{-1}(I - L)^{-1} = d_s f(L + U', \Omega); \end{aligned}$$

$$\bar{g}(\bar{L}+\bar{U}',\bar{X},\mathrm{T}) = [(\bar{L}+\bar{U}')\bar{X}]/-t = -(\bar{D}\bar{X})/t = -(\bar{T}D\bar{R}^{-1}\bar{X})/t$$
$$= -(tD\bar{R}^{-1}\bar{X})/t = -U^{-1}\bar{R}^{-1}\bar{X};$$

$$\bar{h}(\bar{L}+\bar{U}',\bar{X},\mathrm{T}) = \bar{X}/t = \bar{X}(t^{-1})(I-0)^{-1} = \bar{X}\bar{T}^{-1}(I-L)^{-1}.$$

Induction step: Assume the identities stated above hold for $\bar{L}+\bar{U}'$ decompositions of matrices of order less than n and prove it for those of order up to $2n$. The matrices and vectors are decomposed into blocks less than n in size, labelled as usual:

$$\bar{L}+\bar{U}' = \begin{pmatrix} \bar{L}_{nw}+\bar{U}'_{ne} & \bar{E} \\ \bar{W} & \bar{L}_{se}+\bar{U}'_{se} \end{pmatrix};$$

$$\Delta = \langle \Delta_n, \Delta_s \rangle; \quad \Omega = \langle \Omega_n, \Omega_s \rangle; \quad \mathrm{T} = \langle \mathrm{T}_n, \mathrm{T}_s \rangle.$$

Δ and T have the same content as $\bar{R}$ and (but for reversed signs) $\bar{T}$ and decompose similarly:

$$\bar{R} = \begin{pmatrix} \bar{R}_n & 0 \\ 0 & \bar{R}_s \end{pmatrix}; \qquad \bar{T} = \begin{pmatrix} \bar{T}_n & 0 \\ 0 & \bar{T}_s \end{pmatrix},$$

and the northernmost element of Δ_s (or $\bar{R}_s$) is labelled d_n in the definition of $\bar{f}$:

$$\bar{f}(d_s, \bar{L}+\bar{U}', \Delta, \Omega, \mathrm{T})$$
$$= \begin{pmatrix} \frac{d_s\bar{F}+\bar{B}}{d_n} & \bar{H} \\ \bar{J} & \bar{K} \end{pmatrix}$$
$$= \begin{pmatrix} \frac{d_s\bar{f}(d_n,\bar{L}_{nw}+\bar{U}'_{nw},\Delta_n,\Omega_n,\mathrm{T}_n)+\bar{g}(\bar{L}_{nw}+\bar{U}'_{nw},d_n\bar{E}\bar{J},\mathrm{T}_n)}{d_n} & \bar{g}(\bar{L}_{nw}+\bar{U}'_{nw},\bar{E}\bar{K},\mathrm{T}_n) \\ \bar{h}(\bar{L}_{nw}+\bar{U}'_{nw},\bar{K}\bar{W},\mathrm{T}_n) & \bar{f}(d_s,\bar{L}_{se}+\bar{U}'_{se},\Delta_s,\Omega_s,\mathrm{T}_s) \end{pmatrix}$$
$$= d_s \begin{pmatrix} U_{nw}^{-1}[I-\bar{R}_n^{-1}\bar{E}U_{se}^{-1}(I-L_{se})^{-1}\bar{W}\bar{T}_n^{-1}](I-L_{nw})^{-1} & -U_{nw}^{-1}\bar{R}_n^{-1}\bar{E}U_{se}^{-1}(I-L_{se})^{-1} \\ U_{se}^{-1}(I-L_{se})^{-1}\bar{W}\bar{T}_n^{-1}(I-L_{nw})^{-1} & U_{se}^{-1}(I-L_{se})^{-1} \end{pmatrix}$$
$$= d_s \begin{pmatrix} U_{nw}^{-1} & -U_{nw}^{-1}EU_{se}^{-1} \\ 0 & U_{se}^{-1} \end{pmatrix} \begin{pmatrix} (I-L_{nw})^{-1} & 0 \\ (I-L_{se})^{-1}W(I-L_{nw})^{-1} & (I-L_{se})^{-1} \end{pmatrix}.$$

The third line here results from applying the inductive hypotheses and distributing out the scalars: d_s and d_n. $-W = -\bar{W}\bar{T}_n^{-1}$ and $E = \bar{R}_n^{-1}\bar{E}$ because $\bar{L} = L\bar{T}$ and $\bar{U} = \bar{R}U$. The inductive steps for $\bar{g}$ and $\bar{h}$ are left as exercises. □

Corollary 4.3. *The partial result $\bar{B}$ in the definition of $\bar{f}$ in Algorithm* 4.3 *can alternatively be computed as $\bar{B} = \bar{h}(\bar{L}_{nw}+\bar{U}'_{nw}, d_n\bar{H}\bar{W}, T_n)$.*

Corollary 4.4. *If $\bar{X} = 0$ in an application of either $\bar{g}$ or $\bar{h}$, then the result is also* 0.

This annihilation, already used in Fig. 6, becomes critical here to permit the zero padding in T, as suggested in Section 4.3. Since $\bar{E}$ and $\bar{W}$ must be 0 relative to that

internal padding, a zero entry in T never becomes a divisor; it stands as a cheap sentinel on correctness.

Theorem 4.3. *If* $d, \bar{L} + \bar{U}', \Lambda, \Omega, \mathrm{T}$ *all result from an* $\bar{L} + \bar{U}'$ *decomposition of an integer matrix* $\bar{A}$*, then all divisions that arise in applying* $\bar{f}, \bar{g}$*, and* $\bar{h}$ *are exact.*

Proof. Also by induction, but from the top-level application of $\bar{f}$ through nested calls to $\bar{f}, \bar{g}, \bar{h}$ to their bases. The inductive hypothesis is that $\bar{f}$ always returns an integer matrix.

Basis: At the top level $\bar{f}$ returns the inverse of $(I-L)U$ multiplied by its determinant d_s, an integer matrix.

Induction step: If the result from an outer invocation of $\bar{f}$ is exact, then its four quadrants, $\bar{G}, \bar{H}, \bar{J}$, and $\bar{K}$, can contain only integers. So the three invocations of $\bar{h}, \bar{g}$, and $\bar{f}$ that yield $\bar{H}, \bar{J}$, and respectively $\bar{K}$ must yield exact elements. The computation of $\bar{F}$ can always be interpreted as if all of the original problem south and east of $\bar{L}_{nw} + \bar{U}'_{nw}$ were trivial; e.g. $\bar{L}_{se} = 0$, $\bar{U}'_{se} = I = \bar{U}_{se}$, and $\bar{E} = 0 = \bar{W}$. In that case, $\det \bar{A}$ would have been d_n and $\bar{F}$ would have appeared as the southeastern nontrivial block in the integer inverse $d_n \bar{U}^{-1}(I - \bar{L})^{-1}$. Thus, $\bar{F}$ and $\bar{G}$ necessarily contain only integers and so $\bar{B} = d_n \bar{G} - d_s \bar{F}$ is also exact.

The divisions in the nested calls to $\bar{g}$ and $\bar{h}$ are all exact, because every one of those quotients appears explicitly as an element of $\bar{B}, \bar{H}$, or $\bar{J}$, all of which are integer matrices. The divisions in the nested calls to $\bar{f}$ are also exact by induction. □

Corollary 4.5. *Algorithm* 4.3 *inverts an integer matrix using integer operations, exclusively.*

4.5. Counts of scalar multiplications and divisions

Counts of (exact) divisions are more important than those of multiplications for the integer algorithms, because extended precision or symbolic algebra requires long-division algorithms. Yet, even these counts are artificial under multiprocessing, where costs of communication dominate [6]. Under those rules it can be better to recompute a scalar result locally than to share one stored remotely. Sharing of such partial results arise from the identity and annihilator axioms in Fig. 2 (e.g. `x+ZeroM = x`), which return a second reference to an operand that may be otherwise referred, even after the operation.

As an example of such alternatives, consider for a moment the computation of the integer-analog to the Shur complement, the last formula of Algorithm 4.1. If we interpret algebraic codes like that of Fig. 2 to evaluate submatrices according to strict arithmetic precedence, then we would traverse $\bar{A}$ once to multiply it by the scalar t, another time to subtract the $\bar{G}\bar{A}$ colonnade-stripe product, and once again to divide by the scalar d_l^k. Such a strategy requires construction and recovery of three matrices or, better, traversing one thrice and updating it in place.

A better strategy is to cast this computation, instead, as a parallel recurrence on $\bar{A}$ with local cache containing $\tilde{G}$, t, d_l^k, and indices to the pivot block.[4] That is, the opportunity to share intermediate submatrices may not be worth the difficulties that precedence-order evaluation causes: repeated traversals of large memory-resident structures, and repeated construction and recycling of intermediate values that go unshared.

OpCount 4.1. *The function $\bar{f}$ can be applied to an $n \times n$ triangulation within $3n^3/4 - n^2/3$ multiplications.*

The increase in the coefficient from OpCount 2.8 is due to a second invocation of $\bar{g}$ from $\bar{f}$, necessary to preserve integral partial results.

OpCount 4.2. *Algorithm* 2.6 *inverts an $n \times n$ matrix within $\frac{13}{12}n^3 - 2n^2/3$ multiplications, exclusive of permutations.*

It is usual [2] to characterize algorithms of this genre by counting divisions with the following results:

OpCount 4.3. *Algorithm* 4.2, *eliminating 1×1 blocks, decomposes $n \times n$ integer matrices within $n^3/3 - n^2/2 + \mathrm{O}(n)$ integer divisions.*

OpCount 4.4. *Algorithm* 4.2, *eliminating 2×2 blocks, decomposes $n \times n$ integer matrices within $n^3/6 - n^2/2 + \mathrm{O}(n)$ integer divisions.*

OpCount 4.5. *Algorithm* 4.2, *eliminating 4×4 blocks, decomposes $n \times n$ integer matrices within $n^3/12 - n^2/2 + \mathrm{O}(n)$ integer divisions.*

The count of divisions above is coupled to the number of (scalar) Shur complements. Except at 1×1 elimination steps, there is one more to compute the cumulative determinant (elements of Δ) at each step.

OpCount 4.6. *If the $n \times n$ matrices $\bar{L} + \bar{U}'$, $\bar{X}$ result from scalar eliminations, then the functions $\bar{g}$ and $\bar{h}$ can each be applied within n^2 divisions.*

OpCount 4.7. *The function $\bar{f}$ can be applied to an $n \times n$ triangulation computed by eliminating 1×1 blocks with $2n(n-1)$ divisions.*

OpCount 4.8. *The function $\bar{f}$ can be applied to an $n \times n$ triangulation computed by eliminating 2×2 and larger blocks within $2n(n-2)$ divisions.*

[4] Since HASKELL's semantics is lazy, it would likely order these arithmetic operations this way in any event; other languages would not. We still need better compilers that would deliver the efficiency of the parallel recurrence from functional code, eliding the context changes that are usual to lazy languages.

The divisions during integer triangulation dominate these measures. Under undulant pivoting it will be difficult to predict just how many pivots of each size will be chosen, but the direct relationship between the distribution of those choices and the effective coefficient of n^3 makes larger pivot blocks desirable. Moreover, good pivoting will also constrain the magnitudes of the operands to later divisions (Section 3.3.3) in order to reduce their costs.

OpCount 4.9. *The total division count during inversion of an $n \times n$ non-singular integer matrix, eliminating 4×4 blocks at each step, is $n^3/12 + 3n^2/2 + O(n)$.*

5. Ordering and collapsing the residue

> Prolonged contact with the computer turns mathematicians into clerks and vice versa. [18, ¶80]

Present implementations [3, 14] of these algorithms do not order the basis of the vector space before elimination. The remarks in this section, therefore, are speculative.

A good preordering algorithm decomposes the domain into nearly independent subdomains, assembling zeros into off-diagonal blocks [9]. Under block-oriented matrix representations, moreover, it should try to compact either zeroes or non-zeroes within each block.

In the case of the quadtree representation, these blocks are the subtrees. To enable such an ordering, the padding mentioned in the first paragraph of Section 1.1 is best permuted into the interior of the matrix before any elimination begins. That is, a preordering yields a factoring of the permutations $P = P_1 P_0$; $Q_0 Q_1 = Q$ where P_0 and Q_0 are the permutations resulting from preordering the problem, and P_1 and Q_1 were those resulting from pivoting. Then $A_0 = P_0 A Q_0$ in Algorithms 2.1 and 4.1, P_1 and Q_1 are used in Algorithms 2.2 and 4.2, but Algorithms 2.4–2.6 and 4.3 still unwind the rearrangements in a single step using P and Q.

As elimination proceeds under complete pivoting, large stripes and colonnades will have been stripped from the core of original matrix, leaving spindly trees as residue. After half the matrix has been eliminated, the residual problem can be collapsed to a matrix of half the original order. Under the quadtree representation, such a rearrangement can reintroduce larger pivot blocks, as well as squeezing a level from the tree and its subsequent manipulation.

Several collapses may be appropriate to prune successive trees from larger problems; each collapse constitutes a further factoring of P_1 and Q_1, above. As before, all the permutations are unwound in one step.

It may be appropriate to include a reordering in each collapse, as well, so that a single permutation shrinks both the height of the tree and the number of nodes in it, compressing its internal structure into more useful blocks. This is particularly appropriate when there is a significant cost in distributing and synchronizing a permutation across remote memories.

In this context, the padding of Section 3.2 can be characterized as a postordering on the elimination algorithm.

6. Conclusions

> If we believe in data structures, we must believe in independent (hence simultaneous) processing. For why else would we collect items within a structure? Why do we tolerate languages that give us the one without the other? [18, ¶68]

6.1. History

This work began as an exercise in applying lessons from functional programming to a problem that was so well refined that customized hardware (pipelined processors) had already found a large market. It was a test – now a confirmation – of a personally held thesis that functional (or applicative) style is necessary to realize the promise of general-purpose parallel programming in order to liberate the mortal programmer from overspecifying the sequence of detailed instructions. No sequentiality is implied there, aside from the dependence (strictness) of a function's results on its arguments. Moreover, that style squeezes out unnecessary synchronization; the algebra within a program, like the identities and annihilators in Fig. 2, becomes visible and algebraic manipulation of a program is safe.

An early disappointment was that a matrix problem, if structured as a list of rows or columns, did not decompose naturally into just a few subproblems; a problem of order n yields n subproblems, rather than two, four, or sixteen. This situation, creating a burden for the scheduler, was inconsistent with my experience in functional programming. With more processors, a scheduler ought to divide-and-conquer, descending breadth-first into a tree of small degree and scheduling available processors on subproblems from deeper levels. The quadtree representation of Section 1 not only met this notion, but also yielded a remarkable unification of sparse and non-sparse algorithms (cf. Table 1).

Another insight was that repeated row or column interchanges are so difficult in the quadtree structure that they had to be deferred and accumulated, to be done once. This constraint resolved itself surprisingly easily, and its resolution exposed a cost buried in tradition. Although permutations may be cheap on uniprocessors, permuting shared data on a multiprocessor requires communication with sharing processors. Moreover, row or column interchange raises difficulties under other block representations, as well; so it may be best to defer permutations – even on uniprocessors – in favor of a blocked matrix structure. Finally, by delaying permutations, they can be coupled to blocking of the matrix (Section 3.2) or to restructuring of the underlying basis (Section 5).

Several kinds of undulant-block pivoting have now been tested on quadtrees using the following global strategy: compute local measures to decorate a submatrix, and run a tournament to bubble the best choice up the quadtree as each Shur complement is

computed. Different local attributes identify a favorable candidate for a pivot block: large magnitude for floating-point stability, small magnitude for exact arithmetic, large size for parallelism, and minimal fill-in predictions to sustain sparseness. Sketched in Section 3.3, this work is ongoing.

The algorithms described here have now been implemented in various languages and run on various computers, both uniprocessors and a multiprocessor. The languages are C, SCHEME, and HASKELL [14]; the computers include BBN Butterflies [3], a NeXT, SGIs, Suns, and Macintoshes. Test data has been taken from the Harwell–Boeing collection [10]. Performance is favorable, within the ability of each implementation to mimic an idealized environment.

The leaf nodes may be the 1×1 scalars as defined here, or conventional arrays sized to fit in a line of cache. In the latter case, the decoration on the compact array may be carried alongside it or recomputed from the array locally, as needed. It is also possible to augment, say, a 16×16 sequential FORTRAN array with decorations at all 85 of its interior quadtree nodes by augmenting indexing to an Ahnentafel scheme.

The idealized environment has been a parallel processor with a heap-based memory, and language support to recycle uniquely referenced nodes efficiently. It is required that the use and recycling of such nodes *not* mandate global garbage-collection. Indeed, these algorithms have also been used quite successfully to test a hardware-based reference-counting memory [24] with great success; all storage is recycled in real time and garbage collection is never needed.

George [16] observed that block pivoting offers a middle ground between the complicated programming necessary for sparse matrix techniques, and the fill-in that results from straightforward code using a band or band-like ordering. Undulant block-sizing is developed here in an effort also to expand his unified approach from sparse/banded problems to elimination of blocks of varying order, to exact arithmetic, and to multiprocessing.

6.2. Comparisons

Not so long before the serial addressing of a FORTRAN array fixed our attention on row/column operations, undulant-block decomposition was recognized [11] as an important way to divide-and-conquer matrix problems. More recently, block organization re-emerged [17] for sparse matrices [16] and as a tactic to improve locality, a problem that manifested itself earlier as page faults [13], and lately as block transfers among local memories of a multiprocessor [15].

The idea of distributing the search across the preceding pivot [22] is not new here [20, p. 154], but it is remarkable that its desertion was not attributed to limitations of architecture (at that time, before RISC), but rather to the shadow that the higher-level programming language cast over it. Contemporary programming styles can better manipulate both functions that return multiple results [12] and tree structures that stash intermediate decorations at internal nodes.

The impact of this distribution on locality in large systems is stark. Not only does it eliminate the extra traversal (communication) usually associated with separate pivoting, but also it admits complete pivoting at the cost of little space. Moreover, if the matrix is sparse and the algebra of zero annihilates blocks from Shur complements, then an internal decoration often remains correct across many eliminations that do not visit it.

This paper shows how to implement undulant-block pivoting in general, and shows that, in particular, it fits the quadtree representation of matrices well. Quadtrees constrain the search for pivot blocks, and admit a speedy heuristic for identifying non-trivial, non-singular pivot blocks of various sizes, as well as a strategy for managing arithmetic on them. They assist the scheduler with divide-and-conquer decomposition of *small* degree that yields `mapping` functionals (including the `zipWith` in this HASKELL code) to identify a few subprocesses of approximately equal load. Thus, they illustrate the desired efficiencies of undulant-block elimination and its related, sequel codes.

A final comment is necessary on the opCounts on the algorithms; they are almost useless characterizations, except to prescribe asymptotes on uniprocessors and to measure the distribution of work. On modern machines, and certainly on multiprocessors, patterns of memory fetch and store will dominate these operation's times. Communication among multiple processors, or between a processor and shared memory, becomes a huge issue. Block decomposition, a relaxed choice of pivot-candidate size, and the distribution of pivoting over elimination, all blend together to enhance parallel behavior, by doing more on each of fewer traversals.

6.3. Challenges

The issues raised by this paper are of two varieties. One, distinctly avoided here is direct comparison of this family of algorithms to others that solve linear systems. It was written, rather, to reach a wider audience than the problem, alone, circumscribes. My presentation stresses the perspective of many fields of computing research, so that it becomes difficult to extract such a comparison from one field, alone.

For instance, how would analysts of algorithms measure Algorithm 2.6 to be superior to Algorithm 2.5? A difference, based on relative locality, certainly exists. How should we measure the complexity of these algorithms? The opCounts offered here should be replaced by something reflecting locality [6] and read-only cache use, but what? Can good implementations – even on uniprocessors using row-major array representation – expose the greater locality of Algorithm 2.6? And can we use the new measures to identify better algorithms for related problems – especially those that we think are already "solved".

Language writers and, especially, compiler writers should provide more of the expressiveness of this HASKELL code (e.g. multiple return-values and multiple-aggregate `maps`) into parallel languages. Compilers should detect uniquely referenced structures (nodes in a tree) during compilation and convert code like this, that appears to avoid

side effects by allocating "new" space, into run-time code that reuses extant structure.

Large memory has access time at least logarithmic in its size, reflecting the tree in its addressing hardware; how should we design both algorithms and machines better to distribute computation into this tree – instead of placing all the processors at its root? Even multiprocessor architects might ponder the communication patterns that arise from the high-level decomposition in this code. For operating systems: how should writes into subtrees be cheaply synchronized across processors? A simple, but drastic, solution is read-only access into a shared tree; when can overwriting be enabled?

Those interested in scientific computation might ponder the hybrid versions of this code, which use the tree decomposition at higher levels, but have leaves that are, themselves, non-trivial FORTRAN arrays. How should the pivot search proceed? Who should distribute the subtrees across memory, the programmer or the 'system' Are row/column permutations still practical under a hybrid representation? Does the complete pivoting proposed here survive in the hybrid? How should an artful *QR* factorization proceed on quadtree matrices?

Can recursive descent into trees be similarly used to embed asynchronous parallelism in other problems on aggregate structures, like those arising in database management? Symbolic algebraists, who do not usually deal with large problems, might also apply the approach of Section 4 to other symbolic problems. Issues of managing storage are particularly important for them, and good solutions ironically may require sacrifice of some newly acquired algebra. For example, we probably should abandon the right identities in Fig. 2 in exchange for compile-time detection of unique references in quadtrees to enhance storage management.

Most importantly, answers to all these questions will depend on responses from colleagues in various fields. For instance, good compilers from the programming-language people will relieve the burden of implementations for scientific computation. It ought not be necessary to hand-code these algorithms for your favorite multiprocessor; compilers are supposed to generate such low-level code.

So this paper addresses many corners of computing research, that should be interacting better with one another. Lasting solutions will not come from one corner, alone; let the conversations begin!

Acknowledgements

I am particularly indebted to S. Kamal Abdali for his introduction to algorithms from symbolic computation and for his remarkable patience in waiting for my crisp formulation of matrix inversion on quadtrees. Thanks also to Tektronix Labs, where this work began and to NSF for supporting related projects. I also appreciate the encouragement and suggestions of many colleagues, particularly Cleve Ashcraft, John Gilbert, and Erich Kaltofen, over several years' development of this material.

References

[1] S.K. Abdali, D.S. Wise, Experiments with quadtree representation of matrices, in: P. Gianni (Ed.), Proc. ISAAC 88, Lecture Notes in Computer Science, Vol. 358, Springer, Berlin, 1989, pp. 96–108.

[2] E.R. Bareiss, Sylvester's identity and multistep integer-preserving Gaussian elimination, Math. Comput. 22 (103) (1968) 565–578.

[3] P. Beckman, Parallel LU decomposition for sparse matrices using quadtrees on a shared-heap multiprocessor, Ph.D. Dissertation, Indiana University, Bloomington, 1993.

[4] F.W. Burton, J.G. Kollias, Comment on 'The explicit quad tree as a structure for computer graphics', Comput. J. 26 (2) (1983) 188.

[5] H.G. Cragon, A historical note on binary tree, SIGARCH Comput. Archit. News 18 (4) (1990) 3.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: a practical model of parallel computation, Commun. ACM 39 (11) (1996) 78–85.

[7] J.W. Demmel, N.J. Higham, Stability of block algorithms with fast Level 3 BLAS, ACM Trans. Math. Softw. 18 (3) (1992) 274–291.

[8] J. Dongarra, J. DuCroz, S. Hammarling, R. Hanson, An extended set of FORTRAN basic linear algebra subprograms, ACM Trans. Math. Softw. 14 (1988) 1–17.

[9] I.S. Duff, A.M. Erisman, J.K. Reid, Direct Methods for Sparse Matrices, Clarendon Press, Oxford, 1989.

[10] I.S. Duff, R.G. Grimes, J.G. Lewis, Sparse matrix test problems, ACM Trans. Math. Softw. 15 (1) (1989) 1–14.

[11] V.N. Faddeeva, Computational Methods of Linear Algebra, Dover, New York, 1959. Translated from Russian, originally published Moscow, 1950.

[12] J. Fasel, P. Hudak, S. Peyton Jones, P. Wadler (Eds.), HASKELL special issue, SIGPLAN Not. 27 (5) (1992).

[13] P.C. Fischer, R.L. Probert, Storage reorganization techniques for matrix computation in a paging environment, Commun. ACM 22 (7) (1979) 405–415.

[14] J. Frens, D.S. Wise, Matrix inversion Using quadtrees implemented in GOFER, Technical Report 433, Computer Science Dept., Indiana University, May 1995.

[15] K.A. Gallivan, R.J. Plemmons, A.H. Sameh, Parallel algorithms for dense linear algebra computations, SIAM Rev. 32 (1) (1990) 54–135.

[16] A. George, On block elimination for sparse linear systems, SIAM J. Numer. Anal. 11 (3) (1974) 585–603.

[17] G.H. Golub, C.F. Van Loan, Matrix Computations, 2nd ed., The Johns Hopkins University Press, Baltimore, 1989.

[18] A.J. Perlis, Epigrams on programming, SIGPLAN Not. 17 (9) (1982) 7–13.

[19] H. Samet, Algorithms for the conversion of quadtrees to rasters, IEEE Trans. Pattern Anal. Mach. Intell. PAMI-3(1) (1981) 93–95.

[20] G.W. Stewart, Introduction to Matrix Computations, Academic Press, New York, 1973.

[21] D.S. Wise, Representing matrices as quadtrees for parallel processors (extended abstract), SIGSAM Bull. 18 (3) (1984) 24–25.

[22] D.S. Wise, Parallel decomposition of matrix inversion using quadtrees, Proc. Internat. Conf. on Parallel Processing, IEEE Cat. No. 86CH2355-6, 1986, pp. 92–99.

[23] D.S. Wise, J. Franco, Costs of quadtree representation of non-dense matrices, J. Parallel Distrib. Comput. 9 (3) (1990) 282–296.

[24] D.S. Wise, B. Heck, C. Hess, W. Hunt, E. Ost, Uniprocessor performance of a reference-counting hardware heap, Lisp Symb. Comput. 10 (2) (1997) 159–181.